

© 2021 Rafael Fauza

IMPLEMENTATION AND REMOTE DELIVERY OF EMBEDDED SYSTEMS LABS  
ON THE ARDUINO PLATFORM

BY

RAFAEL FAUZA

THESIS

Submitted in partial fulfillment of the requirements  
for the degree of Master of Science in Computer Science  
in the Graduate College of the  
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Adviser:

Professor Lui Raymond Sha

## ABSTRACT

A knowledge of Embedded Systems empowers a computer scientist to solve problems extending beyond the virtual environment of the computer, into the physical world. Such systems are becoming increasingly ubiquitous and constantly extending to new applications, from smart devices, to self-driving cars, to rockets and spacecraft. While knowledge in this area is continues to grow in importance, most students of Computer Science still do not get experience with Embedded Systems, since this is not part of a standard Computer Science education. In this thesis, we have developed 5 laboratory assignments that cover topics including the required electronics knowledge, and low-level control of registers, timers, interrupts, analog and digital I/O, serial communication and servo motors. We believe that the material covered in these assignments, and explored in detail in this text, will provide students with the necessary knowledge and vocabulary to communicate with dedicated Embedded Engineers in industry, or to explore the area further for their own personal development.

## ACKNOWLEDGMENTS

I would like to dedicate this work to my parents for their continued unwavering support, and to my advisor, Dr. Lui Sha, without whom this work would not have been completed.

## TABLE OF CONTENTS

|            |                                                                     |    |
|------------|---------------------------------------------------------------------|----|
| CHAPTER 1  | INTRODUCTION . . . . .                                              | 1  |
| 1.1        | Motivation . . . . .                                                | 1  |
| 1.2        | Learning Objectives and Lab Design . . . . .                        | 1  |
| 1.3        | Hardware Requirements . . . . .                                     | 2  |
| 1.4        | Comparison with Previous Implementation . . . . .                   | 2  |
| CHAPTER 2  | EMBEDDED SYSTEMS AND THE ARDUINO PLATFORM . . . . .                 | 4  |
| 2.1        | What is an Embedded System . . . . .                                | 4  |
| 2.2        | Importance . . . . .                                                | 4  |
| 2.3        | Examples and Applications . . . . .                                 | 5  |
| 2.4        | Electronics Knowledge Requirements . . . . .                        | 5  |
| 2.5        | Microcontrollers vs Single-Board Computers . . . . .                | 10 |
| 2.6        | The Arduino Platform . . . . .                                      | 11 |
| 2.7        | Comparisons with the FLEX Board . . . . .                           | 12 |
| CHAPTER 3  | ARDUINO PROGRAMMING . . . . .                                       | 13 |
| 3.1        | The Arduino IDE . . . . .                                           | 13 |
| 3.2        | Arduino C . . . . .                                                 | 14 |
| 3.3        | Program Structure . . . . .                                         | 17 |
| CHAPTER 4  | ASSIGNMENTS IN DETAIL . . . . .                                     | 29 |
| 4.1        | Assignment 1: Setup and Digital I/O . . . . .                       | 29 |
| 4.2        | Assignment 2: Timer Registers and Interrupts . . . . .              | 34 |
| 4.3        | Assignment 3: Analog to Digital Converters and Analog I/O . . . . . | 38 |
| 4.4        | Assignment 4: Joystick and Servo Motor . . . . .                    | 41 |
| 4.5        | Assignment 5: Serial Communication . . . . .                        | 45 |
| CHAPTER 5  | FUTURE WORK . . . . .                                               | 49 |
| CHAPTER 6  | CONCLUSIONS . . . . .                                               | 50 |
| REFERENCES | . . . . .                                                           | 52 |
| APPENDIX A | LABORATORY ASSIGNMENTS . . . . .                                    | 53 |
| A.1        | Assignment 1 Setup and Digital I/O . . . . .                        | 54 |
| A.2        | Assignment 2 Timer Registers and Interrupts . . . . .               | 65 |
| A.3        | Assignment 3 Analog to Digital Converters and Analog I/O . . . . .  | 75 |
| A.4        | Assignment 4 Joystick and Servo Motor . . . . .                     | 83 |
| A.5        | Assignment 5 Serial Communication . . . . .                         | 94 |

## CHAPTER 1: INTRODUCTION

### 1.1 MOTIVATION

Embedded Systems is the area of Computer Science which bridges Programming and Electronic Circuits, thus allowing complex physical devices to be controlled by software. A working knowledge of Embedded Systems Programming is a powerful tool that allows a Computer Scientist to solve a whole new paradigm of problems, and enables them to face challenges and create solutions that extend beyond the virtual environment of the computer. Because Embedded Programming verges both the Programming and the Electronics areas, a knowledge of both is required for this purpose. The necessary knowledge of Electronics is not advanced, but it is of particular note since it is generally not covered in other Computer Science courses. The types of circuits involved here also differ from the digital logic circuits normally studied in Computer Architecture classes, though those can certainly be used in the external electronics the microcontroller interfaces with. In the learning of Embedded Programming, there is no substitute for hands-on experience and experimentation for the understanding and long-term retention of the material, particularly for the electronics component [1].

### 1.2 LEARNING OBJECTIVES AND LAB DESIGN

In this class, we aim to provide the students with introductory knowledge of, experience with, and vocabulary on Embedded Programming and Realtime Systems, which will enable them to successfully communicate with dedicated embedded and electronic engineers who may be working on the same team on complex projects in industry. We also aim to give them the tools and resources to pursue further study in the area if they wish to do so. To provide students with the hands-on experience that will allow them to enter the world of Embedded Systems, we divided the class into five physical laboratory assignments, covering all the same topics as the previous incarnations of this class, but using more current and widely used hardware. Additional to previous years, this semester's labs also include basic electronics knowledge required to properly setup and operate the physical components in the lab kits. The first lab guides students through the basic electronics theory they will need, as well as providing a detailed guide on how to put together their components and setup the lab. It also covers the basics of connecting the Arduino to a computer and programming it. The second lab focuses on timers, interrupts and registers, including the different timers available

on the Arduino, how to access their respective registers and prescalers, polling, hardware and software interrupts. The third lab covers the topics of analog to digital conversion and analog I/O. Students learn about voltage dividers in order to understand how to generate and vary analog signals, and how to generate analog signals using pulse-width modulation. In lab 4, we apply the pulse-width modulation signals learned in lab 3 to actuate a servo motor, controlled by an analog signal sensed on a separate pin. Finally, in lab 5 the students learn about serial communication, by using the Arduino board to communicate with a Python script running on their computer.

### 1.3 HARDWARE REQUIREMENTS

The new hardware must be capable of driving an LCD display for interaction with the user, providing enough current to drive a DC or Servo motor, it must have sufficient digital and analog input and output pins, analog to digital conversion, servo motor control, serial communication, and it needs to allow the handling of low-level features like interrupts and direct register access. It must also provide connectivity and capability for further development of the labs when more sophisticated peripheral hardware is available, when physical use of the laboratory rooms and resources at the University recommences.

Since the lab assignments had to be completed remotely by each student individually, due to COVID-19 restrictions in the Spring semester of 2021, the hardware had the additional requirement of being inexpensive and accessible, without compromising any of the connectivity or capabilities of the board, and it must be used in the real world by industry and other projects. The Arduino platform beautifully covers all the above requirements, while also providing access to low-level features for advanced users, a plethora of software libraries, tutorials, forums and support from a devoted user community.

### 1.4 COMPARISON WITH PREVIOUS IMPLEMENTATION

The previous implementation of the laboratories for CS431 used an Erika Enterprise FLEX board [2], containing a dsPIC33 microcontroller [3]. It was programmed in C, and it requires direct manipulation of registers. The registers were accessed through the use of macros and manipulated with bitwise operations. This included input/output pins, in addition to timers, interrupt flags, etc. The name of each macro was specific to the platform and required reference to the technical specifications of the chip. Part of the purpose of these labs was to

get students used to referencing the technical materials of the microchips they might come to work with, as this is routinely done when programming chips at a low-level. In more modern systems, however, such as the Arduino or Raspberry Pi, less time is required to be spent reading through processor-specific technical documentation, and more time may be spent learning the capabilities of the board and completing projects. This is especially useful when building solutions to real problems outside of class. The Arduino is very widely used and I believe it to be a more adequate experience for a first course in Embedded Programming today. As the Arduino platform still uses a microcontroller (ATmega328P) at its core, it is also capable of low-level, direct chip programming as done in the previous labs, so this can still be pursued by interested students independently or offered in higher level classes beyond this introductory course.

Having been delivered in-person, the previous labs had the opportunity to access shared University hardware, which allowed for more complex and expensive setups. They used the "Amazing Ball System" [4], where a resistive touchscreen actuated by two servo motors sensed the position of a metal ball, and students wrote software to balance this ball at the center of the platform. This was a very good setup that we unfortunately did not have access to in the Spring of 2021, so we replaced this component of the labs with variable resistors and the servo motor included in the Arduino kit we were able to provide to the students. For future semesters should attempt to reintegrate this hardware into the Arduino labs, as the current lab has no substitute for the application of control theory that the Amazing Ball platform provided.

## CHAPTER 2: EMBEDDED SYSTEMS AND THE ARDUINO PLATFORM

### 2.1 WHAT IS AN EMBEDDED SYSTEM

Embedded Systems have at their center a microcontroller chip, surrounded by support hardware on the same printed circuit board. A microcontroller is a system-on-a-chip, which is an entire computer in a single piece of silicon. This includes CPU cores, memory and permanent storage. In contrast to regular computers, microcontrollers have limited computational and storage capacity, but provide pins for analog and digital external connections. These external interface pins of the microcontrollers can be used as inputs and outputs, which allows the software running on it to sense and command external circuitry. Input pins sense voltages, and these values are written to internal registers by the operating system, and subsequently used by the software. Digital output pins can be programmed to turn on and off (i.e. the pin can be internally connected to the power supply voltage or to ground), thus turning a connected circuit on or off.

The support hardware on these boards can include voltage regulators and capacitors to provide the microcontroller with adequate power, crystal oscillators to provide the clock signals for the CPU and any available timers, physical pins to more easily connect the microcontroller to external circuits, a USB chip and connector to interface with a desktop computer for programming and other peripherals.

### 2.2 IMPORTANCE

Microcontrollers allow for software to directly sense and control electronic devices and components. This enables for the construction and development of extremely more complex technologies that would be infeasible, and in some cases impossible, to build using electronic circuits alone. The cost of building software is insignificant when compared to building equivalent electronic control circuitry, and the cost and effort involved in modifying or debugging hardware can be monumental, while changing code is essentially free. Software can also be distributed, where many independent units communicate and collaborate, or offloaded to a more powerful machine, where simple, low-power and inexpensive end-units can perform complex behaviors well beyond their computational capacity.

With the continued reduction in transistor feature sizes and the reduction in power con-

sumption and cost, increasingly powerful computer and communication devices are being integrated into wearables, medical implants and other applications, continually expanding the capabilities of the devices that can be created and the problems that can be solved.

## 2.3 EXAMPLES AND APPLICATIONS

Microcontrollers are used in most modern technologies we use today. From wearable devices, to manufacturing machinery, to cars, planes, rockets, medical equipment, all the way to the most cutting-edge space hardware. It's hard to think of modern systems that do not rely on microcontrollers. Modern cars, for example, can have dozens of microcontrollers monitoring and controlling all of their systems. Self-driving cars, in particular rely on an even larger suite of sensors and actuators, and heavy computational capacity. Medical equipment such as electrocardiograms, magnetic resonance scanners, pacemakers, and even remote or micro surgery, where miniaturized instruments perform intricate surgery too complex for human hands, all rely heavily on microcontrollers. Robotics use microcontrollers to interface software and actuators, giving complex behavior to machines. Modern planes, rockets and satellites use microcontrollers for their avionics control.

## 2.4 ELECTRONICS KNOWLEDGE REQUIREMENTS

Basic analog electronics knowledge is required to operate a microcontroller board and peripheral circuits. All the topics mentioned in section 2.4 are covered in the labs (see Appendix).

### 2.4.1 Voltage, Resistance, Current and Wattage

Students must understand the relationship between voltage, resistance, current and wattage in order to keep the components within specs. The components used have a rating of 1/4 watts, and the Arduino Uno board can deliver a maximum of 150 milliAmperes, and a single pin can source no more than 20 milliAmperes [5].

Ohm's Law: Voltage = Current x Resistance:

$$V = I * R \tag{2.1}$$

Watts = Voltage x Current:

$$P = V * I = R * I^2 \quad (2.2)$$

They must also learn how to read resistor markings so they can choose the proper resistance to protect their circuits.

| RESISTOR COLOR CODE |        |        |                            |                          |           |
|---------------------|--------|--------|----------------------------|--------------------------|-----------|
| 5-Band-Resistor     |        |        | 234*100kΩ = 23.4MΩ @ 0.25% |                          |           |
| Color               | Band 1 | Band 2 | Band 3                     | Multiplic.               | Tolerance |
| Black               | 0      | 0      | 0                          | 10 <sup>0</sup> (1Ω)     |           |
| Brown               | 1      | 1      | 1                          | 10 <sup>1</sup> (10Ω)    | ± 1%      |
| Red                 | 2      | 2      | 2                          | 10 <sup>2</sup> (100Ω)   | ± 2%      |
| Orange              | 3      | 3      | 3                          | 10 <sup>3</sup> (1kΩ)    |           |
| Yellow              | 4      | 4      | 4                          | 10 <sup>4</sup> (10kΩ)   |           |
| Green               | 5      | 5      | 5                          | 10 <sup>5</sup> (100kΩ)  | ± 0.5%    |
| Blue                | 6      | 6      | 6                          | 10 <sup>6</sup> (1MΩ)    | ± 0.25%   |
| Purple              | 7      | 7      | 7                          | 10 <sup>7</sup> (10MΩ)   | ± 0.1%    |
| Gray                | 8      | 8      | 8                          | 10 <sup>8</sup> (100MΩ)  | ± 0.05%   |
| White               | 9      | 9      | 9                          | 10 <sup>9</sup> (1GΩ)    |           |
| Gold                |        |        |                            | 10 <sup>-1</sup> (100mΩ) | ± 5%      |
| Silver              |        |        |                            | 10 <sup>-2</sup> (10mΩ)  | ± 10%     |
| 4-Band-Resistor     |        |        | 23*10kΩ = 230kΩ @ 0.5%     |                          |           |

Figure 2.1: Decoding Resistor Markings

Students must also learn about voltage dividers in order to understand how a resistive touchscreen works, as well as for controlling voltages and currents through their circuits. In a simple voltage divider, the voltage measured across a resistor is proportional to the ratio of the measured resistor relative to the total resistance in the path, and the voltage in the power supply.

$$V_{out} = V_{in} * (R_2 / (R_1 + R_2)) \quad (2.2)$$

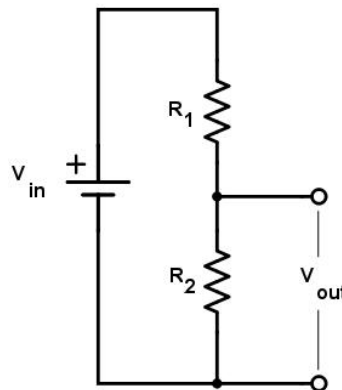


Figure 2.2: Voltage Divider Circuit

### 2.4.2 Electronic Components and Peripherals

The components used in our projects include LEDs, resistors, an LCD display, tactile switches, potentiometers and servo motors. It is important that students understand component polarity for proper use of components such as LEDs. They must be able to read basic schematics in order to build parts of their setup, such as the LCD connection diagram, and also for future work in the area after this class. The setup is built on a breadboard, so in order to use them correctly, a brief overview on breadboard construction is necessary. Switch debouncing is also needed for proper work with microcontrollers and analog circuits. Due to being physical devices, switches bounce very rapidly and briefly when switched on, which the computer can interpret as several on/off events. One way this can be mitigated is by inserting a delay between the time an on event is detected and the next input state change that is registered.

### 2.4.3 Digital Signals

Microcontrollers use both digital and analog signals to sense and control other electronics. Digital signals are read as either "on" or "off". An "off", stored as a 0, value represents 0v (or ground reference), and an "on" value, stored as a 1, represents Vcc (supply voltage). This is true whether that is read on a digital input pin or generated on a digital output pin. Since the digital signal is being extracted from an external analog circuit, there is often variation in the received signals due to interference, noise, out-of-spec components, and a variety of other reasons. This results in signals that are neither ground-reference for "off" bits, nor supply voltage for "on" bits. To address this issue, digital systems will take any voltage above a certain value as a 1 signal, and any value below it as a 0. While the signals that are close to the cutoff voltage still suffer variation, hysteresis is introduced. This is where a small range surrounding the center voltage is considered undefined, and the last read state is retained until the signal crosses the hysteresis threshold again.

As an example, in our 5v Arduino power supply, if we consider 2.5v as the value defining "on" and "off", and use a range of 1v of hysteresis. This results in any value below 2v being considered a 0, and any value above 3v being considered a 1. Whenever the signal oscillates between the 2v and 3v, we disregard that information and consider it to still hold the last read value. So if we previously read a 0 and now we receive 5v, we store a 1. The signal may vary between 2v and 5v and we will still consider it a 1. We will only register a change in (state from 1 to 0) once the signal falls below 2v.

#### 2.4.4 Analog inputs, Analog to Digital Conversion and Quantization

Analog signals can have any value within the operational range, with infinite resolution. Since digital systems are not capable of infinite resolution, analog signals in digital systems must be represented within a number of discrete values between 0v and  $V_{cc}$ . This process is called quantization, and it is unavoidable when converting analog signals for use by a digital system. The resolution of this quantization is dependent on the number of bits available to the Analog-to-Digital Converter (ADC). A 10-bit ADC, such as the ones found on pins A0-A5 of the Arduino Uno, can register values ranging from 0 to 1023 ( $2^{10} = 1024$ ), where 0 represents 0v and 1023 represents 5v. ADC's can be constructed with the use of a resistor ladder (figure 2.3 left), differential amplifiers (figure 2.3 center) and an encoder (figure 2.3 right):

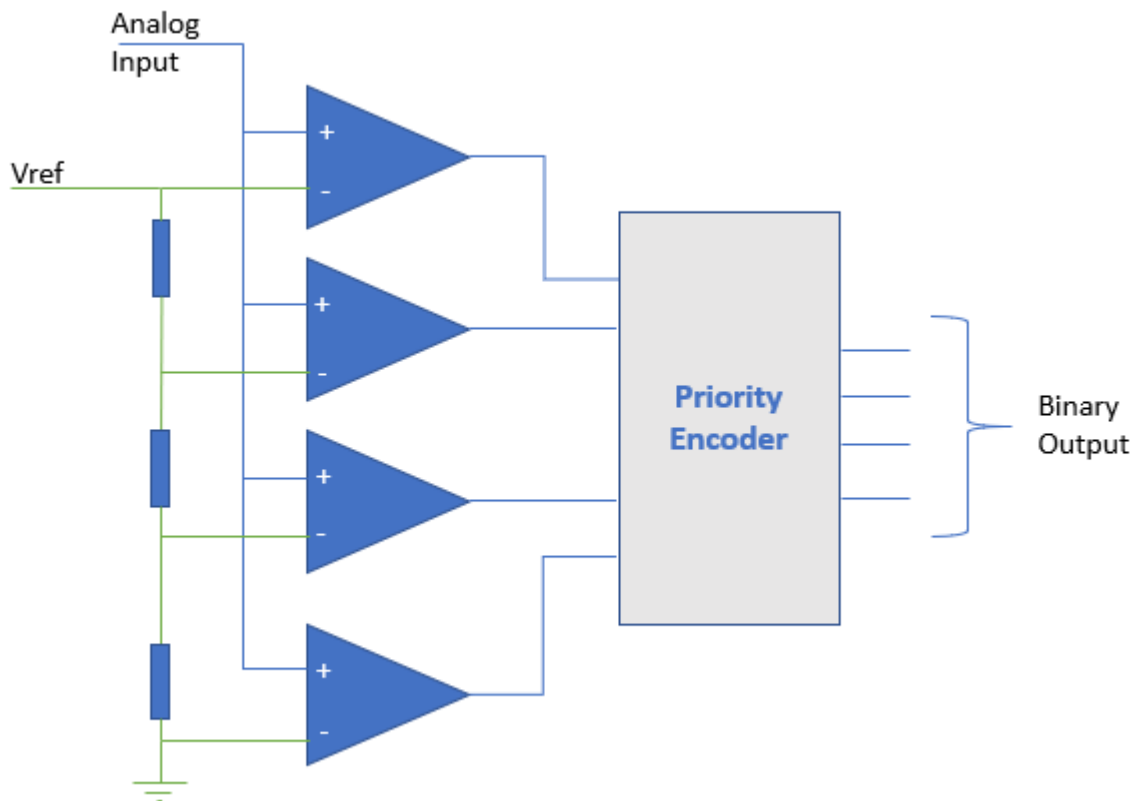


Figure 2.3: 4-bit Analog-to-Digital Converter Circuit

It is not necessary for the students to understand the ADC circuit and how it works, but they must understand its function, and the effects of quantization.

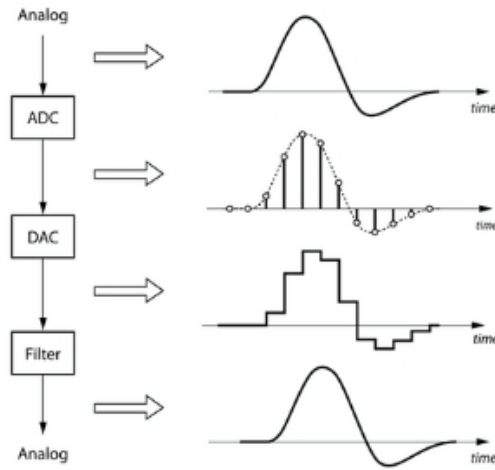


Figure 2.4: Quantization

#### 2.4.5 Pulse-Width Modulation and Digital-to-Analog Conversion

Since the Arduino does not have variable voltage outputs, it uses Pulse-Width Modulation (PWM) to serve a similar purpose. PWM is when a digital signal is rapidly and continuously switched on and off with a controllable duty cycle in order to generate the average voltage desired for the analog output. Since digital signals can only be turned on to maximum supply voltage, or off to 0v, we vary the percentage of time the output signal is held high for each cycle. The frequency of the signal remains unchanged.

PWM is also used to control servo motors. These motors contain extra electronics which allows them to sense and control the angular position of their actuators, where different duty cycle values communicate the angle the motor should rotate to. Unlike DC motors, servos do not turn continuously and have a working range. The servos we use in the labs rotate from  $0^\circ$  to  $180^\circ$ , represented by a duty cycle of 1-2ms respectively, with a frequency of 50Hz.

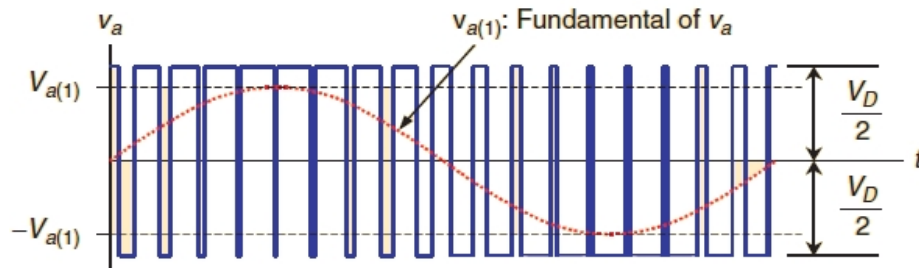


Figure 2.5: PWM Analog Output

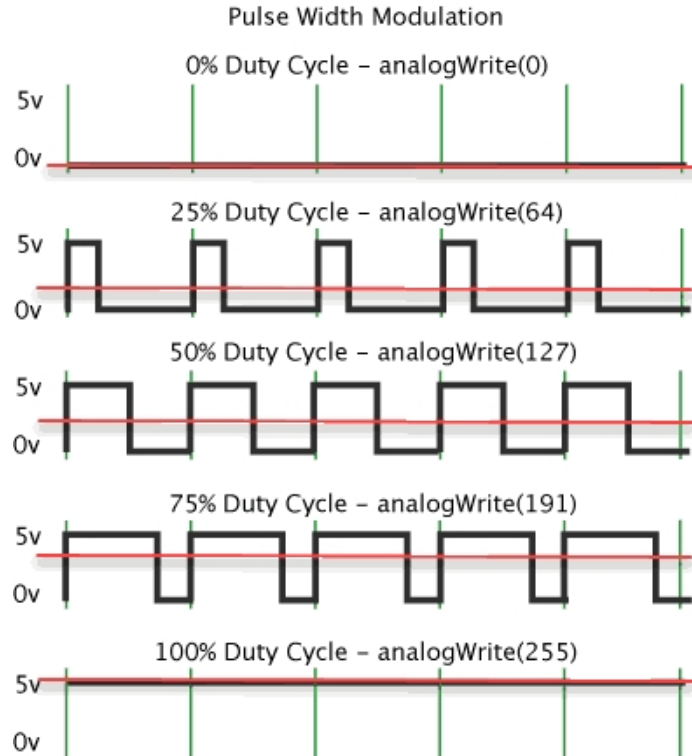


Figure 2.6: Pulse-Width Modulation and Duty Cycle

## 2.5 MICROCONTROLLERS VS SINGLE-BOARD COMPUTERS

Microcontrollers (MCU's) are designed to be small, fast and power-efficient. They usually run minimalist operating systems that bootstrap the main software to run in a loop, in order to minimize latency and maximize responsiveness. Single-Board Computers (SBC), like the RaspberryPi and the LattePanda, are designed to serve all the functions of a desktop computer in a single, efficient package, while also providing I/O connectivity similar to those found on the Arduino. These differ from microcontrollers primarily due to the fact that they are designed to run a desktop operating system. They also employ more powerful and power-hungry hardware, and facilities such as 4k monitor connectivity, WiFi, BlueTooth, Ethernet, USB ports, etc. Some of these SBC's run X86 processors, making direct register access difficult to do. SBC's are a great platform with a wide range of functionality, but are designed for a different purpose than MCU's. SBC's can be used where heavy computation (relative to MCU's) is needed, as for example in computer vision inference on end-devices. Latency is difficult to predict in these SBC's as many processes (including the complex desktop operating system) are running concurrently. MCU's devote their resources to the software being run, so responsiveness is more predictable. MCU's are more appropriate for

applications that require low-power, cost-efficiency and are not computationally heavy.

## 2.6 THE ARDUINO PLATFORM

Arduino is a company that develops and builds open source electronics, mainly focused on microcontroller boards. Their lineup consists of many different boards with varying computational capacity and connectivity for different applications. Because Arduinos are open source, they are inexpensive and accessible, and it has an enormous user base and a wealth of software libraries, tutorials, support, peripheral hardware modules and technical information available.

The board we use in our labs is the Arduino Uno. This board uses an ATmel ATmega328P [6] microcontroller which provides 14 digital I/O pins, 6 of which can produce PWM signals, 6 dedicated analog input pins, 32KB of flash storage, 2KB of SRAM, a clock speed of 16MHz and runs on a 5v power supply. The Uno board provides support circuitry for the microcontroller, including power regulation, 5v and 3.3v rails, USB connectivity for programming from a desktop, a crystal oscillator for the CPU clock, I/O pin extensions for easier connectivity and some other features like built-in LEDs and a reset button.

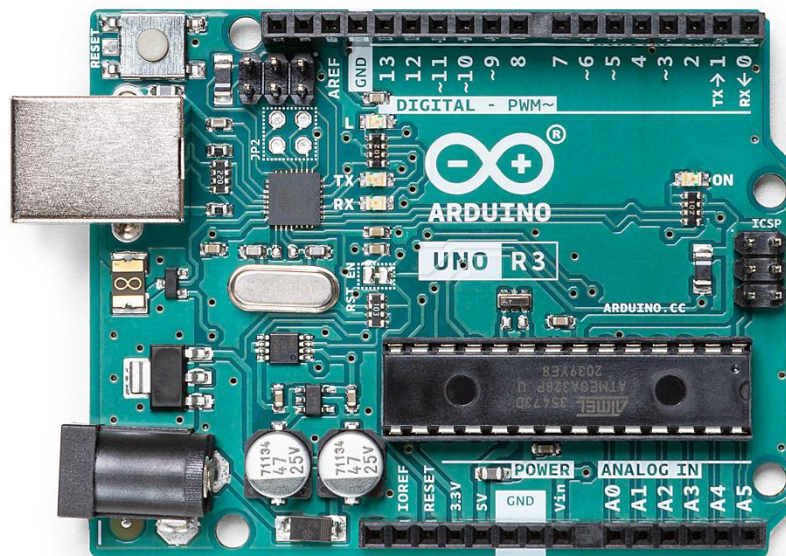


Figure 2.7: The Arduino Board

## 2.7 COMPARISONS WITH THE FLEX BOARD

The FLEX board uses at its center a PIC dsPIC33 [3] microcontroller. This is a more capable chip with up to 85 programmable I/O pins, configurable clock rate with an external oscillator up to 40MHz, 256KB of flash storage, 30KB of SRAM. In the previous labs, we programmed the microcontroller directly through the manipulation of registers and macros which had to be referenced in the technical manual. This is a closer analog to how embedded engineers develop their applications, and the Arduino's ATmega238P is also capable of being programmed in the same way, but this process adds a level of complexity and time to the labs that can be spent building larger projects and exploring more concepts. Some of this complexity may be added back into the new labs once in-person classes resume and a T.A. is physically available while students work on the labs. The Amazing ball kit consisted of several peripheral communication and control boards, and a resistive touchscreen actuated by two servo motors, which balanced a metal ball, as described in section 1.3.

## CHAPTER 3: ARDUINO PROGRAMMING

### 3.1 THE ARDUINO IDE

Microcontrollers traditionally required a dedicated programming device that interfaces between the chip and a computer to upload the desired software. These tended to be expensive devices and required proprietary software. Today, a lot of open source software and hardware designs exist to program these microchips directly.

The Arduino platform takes this a step further by providing an open and free Integrated Development Environment (IDE) [7], and include hardware in their boards that allows the users to connect and program their microcontrollers directly through USB from their desktop or laptop computer. Even single-board computers like the RaspberryPi are capable of programming Arduinos through USB.

The Arduino IDE is designed to be simple and effective. A program meant to run on an Arduino is called a "Sketch". The boards are programmed using Arduino C [8], which is a modified version of the C programming language, extended to include built-in functions for I/O functionality.

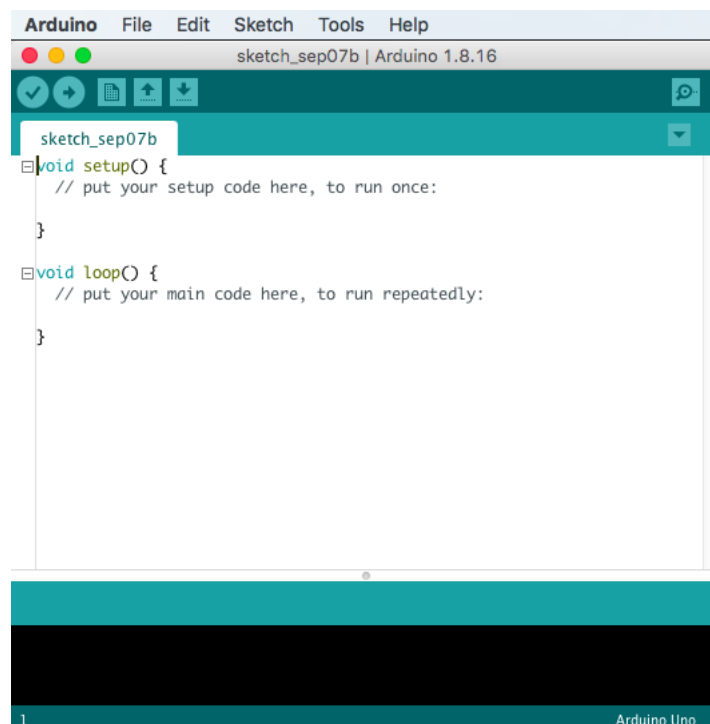


Figure 3.1: The Arduino IDE

## 3.2 ARDUINO C

The Arduino IDE uses a special version of the C programming language for all their Arduino models [8]. This version of C contains much of the same basic syntax and facilities, but is extended to include MCU-specific functionality, such as GPIO (General Purpose Input and Output) pins. Arduino C is also capable of Object-Oriented Programming like C++. Indeed, Arduino is also compatible with C++ libraries [9] and is sometimes referred to as Arduino C++. All the software and hardware provided by Arduino, including software libraries, many sketches and board designs are open source and made to be extensible.

### 3.2.1 Data Types

Microcontrollers have limited memory and storage space available. The ATmega238P used in the Arduino, for example, has only 2KB of RAM. For that reason, Embedded System Programmers have to control the sizes of their variables carefully, and often use bitwise operations (Section 3.3.5) to conserve memory. Added to that, the ATmega238P, like many other microcontrollers, has an 8-bit processor at its core. This means that 8-bit variables can be handled faster and more efficiently by the CPU, and while the MCU provides facilities for using larger variables transparently (as if they were native to the CPU), these require more complex instruction sequences and take longer to process. For variables that are not used in arithmetic operations, the use of unsigned variables is preferred, as signed variables can have erratic behavior as a result of bitwise operations.

Our labs are not demanding enough to require attention to the above, but students should still be aware of these constraints. Following is a comprehensive list of the native data types recognized by the Arduino compiler and their respective sizes on the Uno platform:

### 3.2.2 Arrays vs Strings

Arrays are defined and accessed in the same way as in standard C:

#### Listing 3.1 Arrays

```
char board = 'U', 'n', 'o';  
char platform[7] = "Arduino";
```

Strings are a built-in class as in C++. They are instantiated with the `String()` function and provide a list of methods [8]:

### Listing 3.2 Strings

```
String course = String("CS431");
```

Table 3.1 Data Types

|               |                                                      |
|---------------|------------------------------------------------------|
| array         | 16 bits (pointer to an address)                      |
| bool          | Holds value True or False, but still occupies 8-bits |
| boolean       | Alias for bool                                       |
| byte          | 8-bit unsigned number from 0 to 255                  |
| char          | 8 bits (1 Byte)                                      |
| double        | 32 bits (4 Bytes)                                    |
| float         | 32 bits (4 Bytes)                                    |
| int           | 16 bits (2 Bytes)                                    |
| long          | 32 bits (4 Bytes)                                    |
| short         | 16 bits (2 Bytes)                                    |
| size_t        | variable                                             |
| String        | 16 bits (pointer to an object)                       |
| unsigned char | 8 bits (1 Byte)                                      |
| unsigned int  | 16 bits (2 Bytes)                                    |
| unsigned long | 32 bits (4 Bytes)                                    |
| void          | n/a                                                  |
| word          | 16 bits (2 Bytes)                                    |

### 3.2.3 Constants

The Arduino platform has some built-in constants to abstract the use of binary values of its internal implementation. This allows flexibility, generalization and backwards-compatibility for one unifying platform across a variety of different MCU's.

Table 3.2 Built-in Constants

|              |                                                               |
|--------------|---------------------------------------------------------------|
| HIGH         | a voltage greater than the high threshold (3.0v on 5v boards) |
| LOW          | a voltage lower than the low threshold (1.5v on 5v boards)    |
| INPUT        | sets a digital I/O pin as an input - high impedance state     |
| OUTPUT       | sets a digital I/O pin as an output - low impedance state     |
| INPUT_PULLUP | connects an internal pull-up resistor to the pin              |
| LED_BUILTIN  | 13 (on UNO)                                                   |
| true         | 1                                                             |
| false        | 0                                                             |

### 3.2.4 Numerical Formats

Integers can be defined using decimal, binary, octal or hexadecimal forms. Floating point numbers can make use of a decimal or exponent notation.

Table 3.3 Numerical Types

| Type                | Format                  | Example  |
|---------------------|-------------------------|----------|
| Decimal Integer     | no change               | 1500     |
| Binary Integer      | lead with "B" or "0b"   | 0b110101 |
| Octal Integer       | lead with "0"           | 0147     |
| Hexadecimal Integer | lead with "0x"          | 0x4A7    |
| Decimal Float       | decimal point           | 5.5      |
| Exponent            | "E" denotes power of 10 | 6.32E-4  |
| Exponent            | "e" denotes power of 10 | 1.33e18  |

### 3.2.5 Casting

Variables can be cast as different types using built-in functions. Unsigned int and unsigned long casts still retain the regular C syntax, but other conversions on Arduino should be made with the use of these functions:

Table 3.4 Type Casting

|                 |
|-----------------|
| (unsigned int)  |
| (unsigned long) |
| byte()          |
| char()          |
| float()         |
| int()           |
| long()          |
| word()          |

### 3.3 PROGRAM STRUCTURE

#### 3.3.1 Syntax

The syntax of Arduino C is unchanged from regular C programs. Macro definitions, include directives and global variable declarations are still found at the top of the code and interpreted by the compiler before the required Setup and Loop functions (Section 3.3.2) and any other function definitions. Blocks of code are similarly found between open and close braces. Lines are terminated by semi-colons, and single-line and block comments are preceded by a `"/"` or surrounded by a `"/*" ... "*/"` as in regular C. Includes and macro definitions Also follow traditional C syntax.

E.g.:

```
include <LiquidCrystal.h> //includes library for use of LCD display
define LED1 7 //if LED1 is connected to pin 7, macro improves code readability
```

#### 3.3.2 Setup and Loop

Different from regular programming, Microcontrollers usually run in an infinite loop. The operating system expects all of the code to be run from two functions: `setup()` and `loop()`. Other functions can exist outside of these two, but in order for them to be used they must be called either by `setup()` or by `loop()`.

`setup()` is used to run code that is intended to run only once when the board is started up. This includes initializing variables, pin modes and libraries, attaching interrupts, setting the serial communication baud rate, calling calibration procedures, etc.

After `setup()` is run once, the operating system then calls `loop()`, which will be run repeatedly for the duration of the microcontroller operation. This may sound unintuitive to programmers who are not familiar with microcontroller programming, we are taught to avoid infinite loops at all costs. An embedded environment, however, works differently. The `loop()` function is used to continually monitor and control its inputs and output pins, and to react accordingly. Code is executed conditionally on the values of inputs, variables and timers, and then the process is started again.

### 3.3.3 Control Structures

The control structures recognized by the Arduino compiler are all the expected control structures normally used in regular C, so they will be mentioned here briefly:

Table 3.5 Control Structures

|               |                                                                                |
|---------------|--------------------------------------------------------------------------------|
| for           | repeats a block of code a number of times controlled by a counter              |
| while         | repeats a block of code while a condition remains true                         |
| do...while    | same as while, but first iteration is executed irrespective of condition       |
| break         | used inside a for, while or do...while loop to exit out of the loop            |
| continue      | used inside a for, while or do...while loop to jump to the next loop iteration |
| if            | executes a block of code conditionally                                         |
| else          | found together with an if statement. Executes in case if statement is not true |
| goto          | jumps execution to a defined label in the code                                 |
| switch...case | execute one of a series of code blocks, conditional on the value of a variable |
| return        | terminates a function, may return a value to the caller                        |

Table 3.6 Scope and Utilities

|          |                                                                                  |
|----------|----------------------------------------------------------------------------------|
| const    | modifier that makes a variable read-only. Variable scope, referred over define   |
| static   | modifies local variables to retain value between function calls                  |
| volatile | compiler directive, variable may be changed by external factors during execution |

### 3.3.4 Operators

Arduino C makes use of many of the same built-in operators found in C, including Arithmetic, Comparison, Boolean and Bitwise operators. Operators can also be compounded with the assignment operator to set the value of a variable to the result of an operation.

Table 3.7 Arithmetic and Comparison Operators

| Arithmetic Operators    | Comparison Operators          |
|-------------------------|-------------------------------|
| % (remainder)           | != (not equal to)             |
| * (multiplication)      | < (less than)                 |
| + (addition)            | <= (less than or equal to)    |
| - (subtraction)         | == (equal to)                 |
| / (division)            | > (greater than)              |
| = (assignment operator) | >= (greater than or equal to) |

Table 3.8 Boolean and Compound Operators

| Boolean Operators     | Compound Operators                         |
|-----------------------|--------------------------------------------|
| ! (logical not)       | %= (set variable equal to remainder)       |
| && (logical and)      | *= (set variable equal to the product)     |
| ( <i>logical or</i> ) | ++ (increment)                             |
|                       | += (set variable equal to the sum)         |
|                       | -- (decrement)                             |
|                       | -= (set variable equal to the subtraction) |

### 3.3.5 Bitwise Operations

As mentioned previously, due to the memory constraint of microcontrollers, bitwise operations are often used. Many values used in microcontroller operations, such as the state of a pin, can be expressed in a single bit, but the smallest variable available to the programmer in Arduino C is a Byte. To make efficient use of memory, a 1-Byte variable can hold 8 individual 1-bit values, and each 1-bit value can be accessed and manipulated using bitwise operations and masks.

### 3.3.6 I/O Pin Registers

Registers are used by MCU chips at low-level to control the states of the I/O pins. In the case of the Arduino, they are 1-Byte variables where each bit corresponds to a specific

I/O pin. Manipulating these registers directly is faster than calling the built-in functions provided by the Arduino platform (Section 3.3.6). It should be noted that pins 0 (RX) and 1 (TX) are used by the UNO board for programming and debugging the MCU and writing to these should generally be avoided [10]

Table 3.9 Bitwise Operators

|                       |                      |
|-----------------------|----------------------|
| <code>&amp;</code>    | bitwise and          |
| <code>^</code>        | bitwise xor          |
| <code>  </code>       | bitwise or           |
| <code>~</code>        | bitwise not          |
| <code>&lt;&lt;</code> | bitshift left        |
| <code>&gt;&gt;</code> | bitshift right       |
| <code>^ =</code>      | compound bitwise xor |
| <code>   =</code>     | compound bitwise or  |
| <code>&amp;=</code>   | compound bitwise and |

The DDR registers control whether the respective pins are set to inputs (1) or outputs (0), and the Port registers store the HIGH/LOW states of each pin. Register B governs digital pins 8-13, register C is assigned to the analog input pins (0-5), and register D controls digital pins 0-7. The Arduino built-in libraries provide macros for accessing these registers.

E.g. As an example, let's say we want to connect a light to pin 4 and a motion sensor to pin 5, and turn the light on whenever the sensor detects motion (in this case assume the sensor outputs a LOW signal when it is active. Many devices follow this format in order to reduce current draw from I/O pins). We then need to set pin 4 as an output and pin 5 as an input, and programmatically set pin 4 to output the inverse signal that we are reading from pin 5:

Listing 3.3 Setting pin 4 state

```
DDRD &= 0b11101111 // set pin 4 direction as output
DDRD || = 0b00100000 // set pin 5 direction as input
//set pin 4's output state opposite to pin 5's input state
```

Listing 3.3 (cont.)

```
PORTD ^ = (((PORTD & 0b00100000) >> 1) ^
(PORTD & 0b00010000)) ^ 0b00010000)
```

Examining the last line in more detail, we extract the state of the pin 5 input and right shift it by one to align with pin 4 in the register:

Listing 3.4 Right shifting register 5

```
(PORTD & 0b00100000) >> 1
```

We then extract the value of pin 4 and XOR it with the above, thereby flipping the bit value we read from pin 5 when the old value of pin 4 was 0. This is necessary to avoid having to use separate operators to set (bitwise OR 1) and reset (bitwise AND 0) the bit:

Listing 3.5 XOR with register 5

```
((PORTD & 0b00100000) >> 1) ^ (PORTD & 0b00010000)
```

We XOR the above with the pin 4 mask and XOR the result with the contents of pin the register. Since all bits other than pin 4 in our result are 0's, this will leave the remainder of the register unchanged, while setting pin 4 as desired:

Listing 3.6 XOR with PORTD

```
PORTD ^ = (((PORTD & 0b00100000) >> 1) ^ (PORTD & 0b00010000))
          ^ 0b00010000)
```

Following is a look at the bit-values in this calculation for illustration, using 4 bits for simplicity, where we can verify the logic:

Extract the register value of Pin 5 and right shift it by one:

Listing 3.7 Right shifting register 5

|            |             |             |
|------------|-------------|-------------|
| Pin 5      | x1xx        | x0xx        |
| Pin 5 Mask | <u>0100</u> | <u>0100</u> |
| AND        | 0100        | 0000        |
| >>         | 0010        | 0000        |

Extract the register value of Pin 4:

Listing 3.8 Get value of pin 4

|            |             |             |
|------------|-------------|-------------|
| Pin 4      | xx1x        | xx0x        |
| Pin 4 Mask | <u>0010</u> | <u>0010</u> |
| AND        | 0010        | 0000        |

Set Pin 4 in PORTD register:

Listing 3.9 Set value of pin 4

|            |             |             |             |             |
|------------|-------------|-------------|-------------|-------------|
| Pin 4      | 0000        | 0000        | 0010        | 0010        |
| >> Pin 5   | <u>0000</u> | <u>0010</u> | <u>0000</u> | <u>0010</u> |
| XOR        | 0000        | 0010        | 0010        | 0000        |
| Pin 4 Mask | <u>0010</u> | <u>0010</u> | <u>0010</u> | <u>0010</u> |
| XOR        | 0010        | 0000        | 0000        | 0010        |
| PORTD      | <u>xx0x</u> | <u>xx0x</u> | <u>xx1x</u> | <u>xx1x</u> |
| XOR        | xx1x        | xx0x        | xx1x        | xx0x        |

### 3.3.7 Timer Registers

A timer is a register inside the MCU chip that autonomously increments as the chip runs. A prescaler is a second counter working in conjunction with the timer, which allows us to increment the timer on multiples of the clock frequency being used (1/8, 1/64, 1/256 or 1/1024 for the 8-bit timers, and 1/8, 1/32, 1/64, 1/128, 1/256 or 1/1024 for the 16-bit timer [4]). The value of the prescaler can be set within the program. The ATmega328P has 3 internal timers with dedicated prescalers. Timers 0 and 2 are 8-bit (28 = 256 values), and timer 1 is 16-bit, which gives us a higher resolution (216 = 65536 values). These registers can be accessed directly as TCNTn (where n represents the timer number). These timers enable the Arduino built-in time functions, as well as generating the PWM signals and keeping them in phase.

### 3.3.8 Built-in Functions

To simplify programming, Arduino C provides a large set of built-in functions, so direct register manipulation (Section 3.3.5.1) is not necessary. This makes the platform more accessible and easier to use, at the cost of some efficiency. As we saw in the previous section,

each digital I/O pin must be set as an input or an output. We can do this more simply by use of the built-in function `pinMode(int pin_num, int mode)`. The values can then be read or set (according to the input or output setting of the pin) using the `digitalRead(int pin)` and `digitalWrite(int pin, int value)` functions. Digital output pins 3, 5, 6, 9, 10 and 11 are also capable of generating PWM signals (Section 2.4.3.3) with the use of the `analogWrite(int pin, int value)` function, where 'value' sets the duty cycle of the PWM signal in steps of 1/256, from always off (0) to always on (255).

The Arduino Uno has 6 10-bit analog input pins. These pins are not capable of generating output, and a 10-bit resolution in 5V operations results in 4.9mV steps. The function `AnalogRead(int pin)` is used to access the value of these ADC's. When these pins are left floating (not connected to anything), the results they read will be erratic and random, due to static noise and other factors.

Table 3.10 Built-in Functions

| <u>I/O</u>                   | <u>Characters</u>                 | <u>Math</u>                |
|------------------------------|-----------------------------------|----------------------------|
| <code>digitalRead()</code>   | <code>isAlpha()</code>            | <code>abs()</code>         |
| <code>digitalWrite()</code>  | <code>isAlphaNumeric()</code>     | <code>constrain()</code>   |
| <code>pinMode()</code>       | <code>isAscii()</code>            | <code>map()</code>         |
| <code>analogRead()</code>    | <code>isControl()</code>          | <code>max()</code>         |
|                              | <code>isDigit()</code>            | 0000                       |
| <b><u>Bits and Bytes</u></b> | <code>isGraph()</code>            | <code>min()</code>         |
| <code>bit()</code>           | <code>isHexadecimalDigit()</code> | <code>pow()</code>         |
| <code>bitClear()</code>      | <code>isLowerCase()</code>        | <code>sq()</code>          |
| <code>bitRead()</code>       | <code>isPrintable()</code>        | <code>sqrt()</code>        |
| <code>bitSet()</code>        | <code>isPunct()</code>            |                            |
| <code>bitWrite()</code>      | <code>isSpace()</code>            | <b><u>Trigonometry</u></b> |
| <code>highByte()</code>      | <code>isUpperCase()</code>        | <code>cos()</code>         |
| <code>lowByte()</code>       | <code>isWhitespace()</code>       | <code>sin()</code>         |
|                              |                                   | <code>tan()</code>         |

Table 3.11 Built-in Utility Functions

### **Random Numbers**

`randomSeed(unsigned long)` initializes the pseudorandom number generator sequence  
`random([min,] max)` initializes the pseudorandom number generator sequence

### **Size**

`sizeof(var)` returns the size of the variable in Bytes

Table 3.11 (cont.)

### **Time**

|                                        |                                                              |
|----------------------------------------|--------------------------------------------------------------|
| <code>delay(int ms)</code>             | pauses program execution by the time given (in milliseconds) |
| <code>delayMicroseconds(int us)</code> | pauses program execution by the time given (in microseconds) |
| <code>micros()</code>                  | returns the time since last power-on in microseconds         |
| <code>millis()</code>                  | returns the time since last power-on in milliseconds         |

### 3.3.9 Interrupts

An interrupt occurs when the execution of the program is halted due to an internal (e.g. clock) or external (e.g. input pin state change) event. These will usually prompt an interrupt handler function to execute to completion before returning to the program where execution was interrupted. Interrupt handlers should be as concise and efficient as possible, as to minimize interference with execution and other interrupts.

Interrupts may be disabled with the function `noInterrupts()`, for use while executing a critical section of code that should not be interrupted, or that uses volatile variables that should not be changed while they are being used. Interrupts should be re-enabled by calling `interrupts()`. Many important background tasks rely on interrupts, so they are enabled by default and should be re-enabled after the critical section is complete.

On the Arduino, while an interrupt handler function is running, other interrupts are queued and handled subsequently. These handler functions can be created by the programmer to handle events in their code, and must take and return no parameters (void). In order to use an interrupt handler, you create your handler function as described above, then 'attach' it using the function: `attachInterrupt(digitalPinToInterrupt(pin), ISR, mode)`

where `pin` is the pin number you want to associate with that handler, `ISR` is the name of the interrupt handler function (Interrupt Service Routine) you created, and `mode` refers to one of four states of the pin triggering the interrupt: `LOW`, `CHANGE`, `RISING`, `FALLING`. We can disassociate the pin with the interrupt by use of the 'detach' function: `detachInterrupt(digitalPinToInterrupt(pin))`

Table 3.12 Interrupt Functions

|                                                                     |                                  |
|---------------------------------------------------------------------|----------------------------------|
| <code>attachInterrupt(digitalPinToInterrupt(pin), ISR, mode)</code> | associates pin event with ISR    |
| <code>detachInterrupt(digitalPinToInterrupt(pin))</code>            | disassociates pin with interrupt |
| <code>noInterrupts()</code>                                         | disables interrupts              |
| <code>interrupts()</code>                                           | re-enables interrupts            |

### 3.3.10 Serial Communication

The ATmega328P MCU provides UART digital communications, which are connected to pins 0 (RX) and 1 (TX) of the Arduino Uno. The Uno board includes an ATmega16U2 USB chip that connects the MCU to the Arduino IDE on a desktop computer, allowing it to be directly programmed via USB, or to receive and transmit data through the serial monitor included in the IDE. Using this, interacting with the program running on the MCU via a facility similar to a terminal is possible. The Arduino community also provides libraries (section 3.2.10) for using any digital I/O pin for digital communication, as well as I2C and SPI. Serial Communication can also be used to connect the Arduino to other serial-capable devices (e.g. 2 Arduino boards can communicate directly).

For regular operations between the IDE serial monitor and the Arduino, the Serial communication must be initialized to the correct baud rate. The baud rate can be selected in the Arduino IDE, and serial communication must be initialized on the board using the 'begin' function same baud rate using: `Serial.begin(const int baudrate);`

Once initialized, we can send information to the serial monitor in the IDE with the use of the print functions, and to receive data entered into the serial monitor with the read functions:

Table 3.13 Serial Communication Functions

|                                    |                                                                     |
|------------------------------------|---------------------------------------------------------------------|
| <code>Serial.print(value)</code>   | can take integers, floats, characters or strings                    |
| <code>Serial.println(value)</code> | includes a newline character                                        |
| <code>Serial.readString()</code>   | returns the string read from the buffer                             |
| <code>Serial.read()</code>         | returns first Byte of available data, or -1 if no data is available |

The Serial library is included by default and does not require a include statement. A full list of the functions available in this library can be found in the Arduino language reference material [11]

### 3.3.11 Hello World

For microcontrollers, the Hello World equivalent is to make an LED blink at a desired frequency. The Arduino IDE provides a "Blink" sketch, as well as many other pre-built sketches, that can be quickly uploaded to the board and visually checked for correct function of the board, MCU and IDE using the built-in LED connected to digital I/O pin 13. This simple sketch can be seen below, it is provided as public domain by the Arduino community and can be seen in more detail on the Arduino website [12].

Listing 3.10 Hello World program

```
void setup() {  
    // initialize digital pin LED_BUILTIN as an output.  
    pinMode(LED_BUILTIN, OUTPUT); }  
  
// the loop function runs over and over again forever  
void loop() {  
    // turn the LED on (HIGH is the voltage level)  
    digitalWrite(LED_BUILTIN, HIGH);  
    delay(1000); // wait for a second  
    // turn the LED off by making the voltage LOW  
    digitalWrite(LED_BUILTIN, LOW);  
    delay(1000); // wait for a second  
}
```

### 3.3.12 Libraries

A truly vast number of open-source libraries exist for the Arduino boards [13], provided and created by the community. These help to make complex functionality much more accessible to programmers. In our labs, we make use of the LiquidCrystal library to operate our LCD screen, and the Servo library to control our servo motor in lab 4. In this section we will briefly discuss the library functions used for the assignments.

### 3.3.13 LiquidCrystal Library

The Liquid Crystal Library requires the creation of a LiquidCrystal object, whose methods we will use to control our LCD screen. When creating this object, we instruct the library on which pins we have connected to the respective inputs on the LCD component.

We then initialize the library functionality by calling the `.begin(int columns, int rows)` method on our lcd object. We can move the cursor with the `.setCursor(int column, int row)` method, clear the screen with the `.clear()` method, and print to it with the `.print()` method. The `.print` method can take a string, character or number. If the LCD is written to without calling `.clear()`, parts of the old text will be overwritten, and the rest will remain on the screen. Overflow text for each line is discarded. The full functionality of the library can be found on the Arduino website [14]

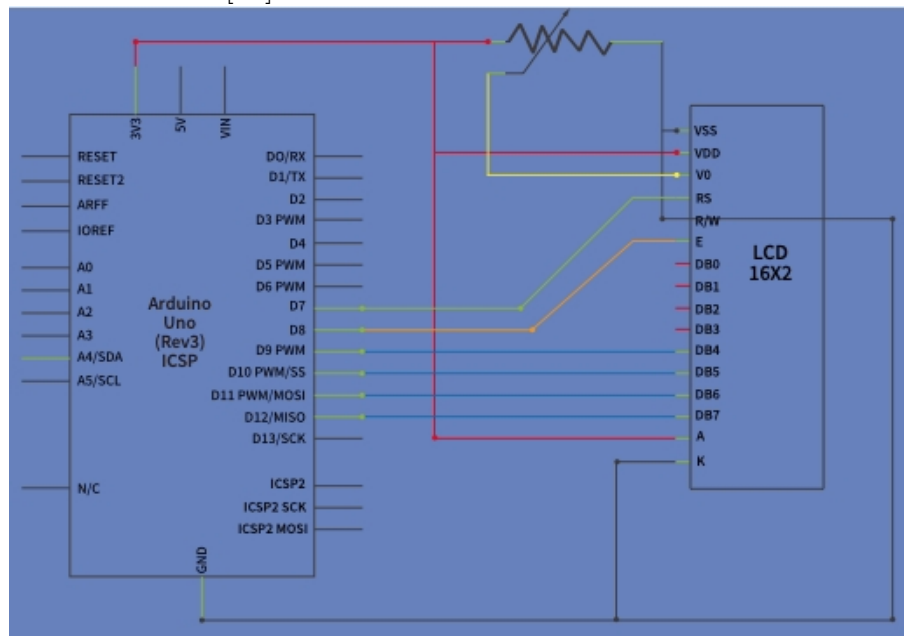


Figure 3.2: LCD Schematic

Listing 3.11 LCD Excerpt from Lab 1

```
include <LiquidCrystal.h> // includes the Liquid Crystal Library
// assign variable names to each pin, corresponding to the LCD component inputs, for
// clarity and debugging
const int rs = 12, en = 11, d4 = 3, d5 = 4, d6 = 5, d7 = 6;
// instantiate our LCD object with the respective pin numbers
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);
lcd.begin(16, 2); // initialize the LCD object
```

### Listing 3.11 (cont.)

```
lcd.clear(); //clear the LCD screen
lcd.setCursor(0, 0); // move cursor to upper left of LCD screen
lcd.print("Welcome to CS431"); //print a string on first line
lcd.setCursor(0, 1); //move cursor to first pixel of second line
lcd.print(0); // print the number 0 on the second line
```

### 3.3.14 Servo Library

The servo library simplifies the process of generating PWM signals to control servo motors. Servos require precise PWM signals, and this library provides support for many different types of servo motors. Servo motors have three wires (power=red, ground=brown/black and signal=yellow), only the signal pin needs to be connected to a digital I/O pin:



Figure 3.3: Servo Motor Connector

The Arduino board sources enough current through its 5v rail to operate our single, small servo motor, but for larger projects, an external power supply should be used to power the motors. For more information on this library and examples of code, see the official Arduino library reference page [15]

### Listing 3.12 Servo Excerpt from Lab 4

```
include <Servo.h>
Servo servo; // instantiate our Servo object
servo.attach(9); // define pin 9 as the PWM control pin for the servo
const int x_in = A0; // use analog pin 0 to store our joystick X coordinate
xIn = analogRead(x_in); // store the X position of the joystick into variable xIn
// calculate the position of the servo in degrees. We divide the reading by 1024 (10-bit
// ADC resolution) and then multiply it by 180° to find the relative position of the
// servo arm
servoPos = (int)((float)xIn/1024)*180);
servo.write(servoPos); // move the servo to the desired angle
```

## CHAPTER 4: ASSIGNMENTS IN DETAIL

In this section, we will look at the requirements and solutions for each Arduino lab assignment as presented in the Spring of 2021. We will focus on the sections of code that should be generated by the students when solving the assignments. Concepts once covered will be omitted in subsequent sections for later labs in the interest of brevity. Each assignment may have more solutions than the one presented. The unabridged code and instruction slides can be found in Appendix A.

### 4.1 ASSIGNMENT 1: SETUP AND DIGITAL I/O

Assignment 1 is meant to guide the students through setting up their microcontroller boards and give them the basic electronics knowledge they will need in order to create microcontroller projects that interact with the physical world.

#### 4.1.1 Learning Objectives

For our labs, we use electronic breadboards to connect our components, as there is no need for soldering, and altering the circuit is very easy. But in order to use these correctly, the students must understand the construction of these devices, and specifically the limitation that the power and ground rails on the board are not connected across the center, so those connections must be made by the student:

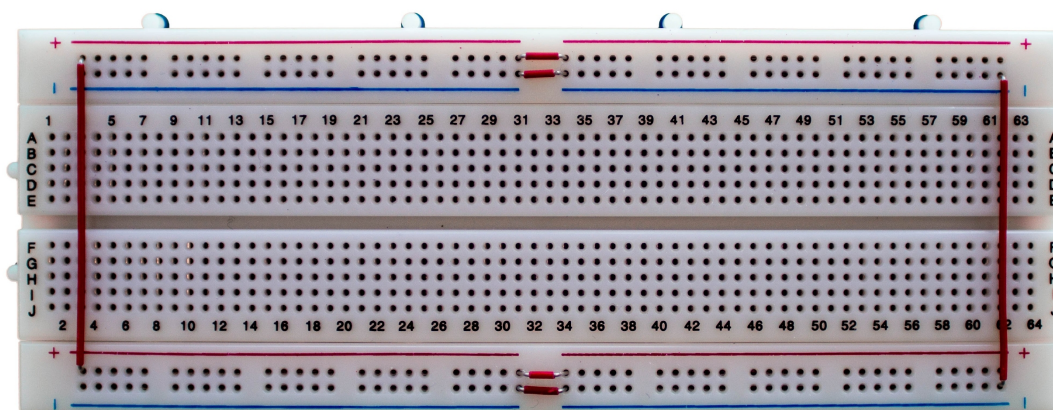


Figure 4.1: Breadboard Rails

They must also understand how to correctly connect their components, as the connections

on each number row are internally wired together for each half of the board, so components must be connected across rows:

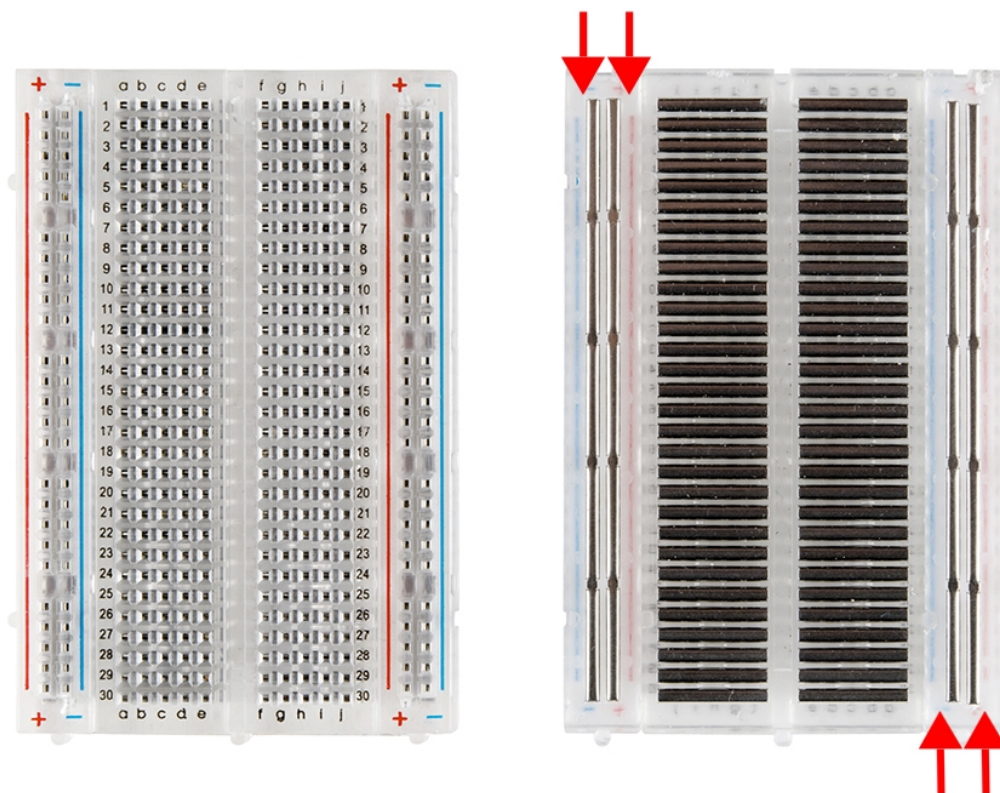


Figure 4.2: Breadboard Internal Construction

Next, the most important concept the students need to understand is that each component has a maximum wattage limitation, and that no component, including the board itself, can be connected directly between power and ground without the proper resistance in the circuit. In order to calculate this resistance, we make use of Ohm's law, the formula for Watts, and a resistor color markings chart (Section 2.4.1).

Students must understand switch contact bouncing (Section 2.4.2) and how to mitigate this physical problem through software debouncing. After learning these important basic concepts, students must assemble their kits, install the Arduino IDE on their computers and program their boards.

#### 4.1.2 Common Errors

The most common error found by students is overlooking the fact that the board's rails are disconnected in the middle. This has the very simple solution of creating these connections

externally (Figure 4.1). A more important error is to neglect to use proper resistance in the circuit. The voltages and currents we are using are low, as the Arduino board can only source 150 milliAmperes at 5 Volts, so these electronics provide no danger to the student. However, the resulting 0.75 Watts is more than the 0.25 Watts the components are rated for, so an LED or low-value resistor can easily be burned out. Additionally, the pins of the Arduino board should not be connected directly to ground, as the unimpeded current draw will stress the microprocessor and can lead to its failure.

#### 4.1.3 Grading Requirements

To complete this assignment, students will need to have their boards and Arduino IDE setup correctly (50%), display their name on the first line of the LCD (10%), and a count debounced switch presses on the second line (10%). They must additionally turn on LED1 when switch 1 is pressed (10%), turn on LED2 when switch 2 is pressed (10%), and LED3 when both switches have the same state (both are closed or open, 10%). The students will need to fill out the provided skeletons of 3 functions: setup, loop and debounce in order to accomplish these requirements.

#### 4.1.4 Global Variables

Since loop() is a function and the Arduino runs this function repeatedly, global variables are required to keep count of button presses, since variables local to the loop() function would go out of scope and reset across iterations. Also the state of the switches will be used by both the loop() and debounce() functions, and thus must be global. Finally we will need to keep track of previous button states and time since last debounce for each one. I chose to keep these in arrays for future extensibility of the debounce function:

Listing 4.1 Global variables

```
int button1State = 0;
int button2State = 0;
unsigned long button1Presses = 0;
int prevButtonStates[2] = 0,0;
unsigned long lastDebounce[2] = 0, 0;
```

#### 4.1.5 Setup Function

In the setup function, the input and output pins' states must be set, corresponding to the correct physical connections between the Arduino pins and the electronic components (switches and LED's in this case). The constants `button1`, `button2`, `LED1`, `LED2` and `LED3` are provided in the code, and reflect the I/O connections in the instructions.

Listing 4.2 Setup function

```
pinMode(button1, INPUT);
pinMode(button2, INPUT);
pinMode(LED1, OUTPUT);
pinMode(LED2, OUTPUT);
pinMode(LED3, OUTPUT);
// set up the LCD's number of columns and rows:
lcd.begin(16, 2);
// Print a message to the LCD.
lcd.setCursor(0, 0);
lcd.print("Welcome to CS431");
lcd.setCursor(0, 1); // set the cursor to column 0, line 1
lcd.print(0);
```

#### 4.1.6 Debounce Function

The provided debounce function solution was designed to be extensible to more pins. Therefore, the function takes a pin number as an argument, and uses a global array (Section 4.4.1) to keep track of the previous pin states and time since last debounce. The latter is important because the method of debouncing chosen for this function involves disregarding state changes on a pin within a given period (50ms) since a state change on that pin was last recorded.

Listing 4.3 Debounce function

```
int state = 0;
if(pin != 10 pin != 2){ //we only accept pins 10 and 2 for this lab
```

#### Listing 4.3 (cont.)

```
    return 0;
}
// convert the pin number to the array index
int button=0;
if(pin==2){
    button = 1;
}
// read the value of the input pin if more than 50ms have passed since last debounce
// else return old value from array
unsigned long currentTime = millis();
if ((currentTime - lastDebounce[button]) > debounceDelay) {
    state = digitalRead(pin);
    lastDebounce[button] = currentTime;
}else{
    state = prevButtonStates[button];
}
return state;
```

#### 4.1.7 Loop Function

We start by getting the debounced state of each tactile switch at the beginning of each loop. We then use a simple logic to increment the counter for button 1 presses if we sense a state change from low to high, update the counter on the LCD and turn on or off the respective LEDs.

#### Listing 4.4 Loop function

```
button1State = debounce(button1);
button2State = debounce(button2);

if(button1State && !prevButtonStates[0]){ // sensed button 1 low to high state change
    digitalWrite(LED1, HIGH); // turn on LED1
    prevButtonStates[0] = 1; // update previous states array for next loop
```

#### Listing 4.4 (cont.)

```
    button1Presses++; // increment button 1 presses counter
    lcd.setCursor(0, 1); // move LCD cursor to first pixel of second line
    lcd.print(button1Presses); // print updated button 1 presses count
}
if(!button1State && prevButtonStates[0]){ // sensed button 1 high to low state change
    prevButtonStates[0] = 0; // update previous states array for next loop
    digitalWrite(LED1, LOW); // turn off LED1
}
if(button2State && !prevButtonStates[1]){ // sensed button 2 low to high state change
    prevButtonStates[1] = 1; // update previous states array for next loop
    digitalWrite(LED2, HIGH); // turn on LED2
}
if(!button2State && prevButtonStates[1]){ // sensed button 1 high to low state change
    prevButtonStates[1] = 0; // update previous states array for next loop
    digitalWrite(LED2, LOW); // turn off LED2
}
// turn on LED3 when buttons 1 and 2 have the same state, and off when they do not
if(button1State==button2State){ digitalWrite(LED3, HIGH);
}else{ digitalWrite(LED3, LOW);
}
```

## 4.2 ASSIGNMENT 2: TIMER REGISTERS AND INTERRUPTS

In this assignment, students use the ATmega328P's timer registers to trigger interrupts, and write their own interrupt handlers to toggle LEDs.

### 4.2.1 Learning Objectives

In this lab, we aim to give students experience directly manipulating registers inside a microcontroller using bitwise operations. Students must learn how to use bitwise operations in C, if not already familiar, and be able to reference the documentation to find the built-in names of the registers and prescalers they would like to manipulate. This is a more powerful and efficient way of programming these devices than is using the built-in Arduino libraries, and is the primary method used in industry. They must also understand the concept of a

prescaler, which controls how often the respective timer register increments, in multiples of the main clock.

The second area this lab covers is interrupts, where the entire operation of the board is halted in order to process a high priority event. These events can be external interrupts, triggered by an I/O pin event (LOW to HIGH, or HIGH to LOW transition), or internal interrupts, triggered by one of the internal timers. Interrupts are handled by Interrupt Service Routines and students must learn how to use and create these.

Finally, the alternate, and less efficient method of handling external events, in the form of polling, is also covered. In polling, the regular program loop running on the microcontroller periodically checks the condition of a pin, and calls a function conditionally on it. This method is not preferred, but students should also learn how to use it.

#### 4.2.2 Common Errors

Prescaler values are a common source of error, as different clock sources can be available, and values must often be converted from frequency to period, so students must be given a clear explanation of these concepts.

#### 4.2.3 Grading Requirements

Students must program their boards to toggle LED2 every 0.5 milliseconds using Timer0's interrupt (25%), and toggle LED3 every second using Timer1's interrupt (25%). They must display the time since last reset on the screen (25%), and reset the counter using an interrupt on pin 2 (25%).

#### 4.2.4 Timer Registers and Prescalers

The documentation provided in the instructions lists all the registers for each timer. Since this concept may be very foreign to many of the students in the class, and since the labs were not attempted in the presence of a T.A., the registers for Timer 0 and prescaler calculation are given. Students must reference the documentation and set the same respective registers for Timer1, as well as calculate and set the prescaler. This is all done inside the setup() function:

#### Listing 4.5 Time registers and prescalers

```
//set timer1 interrupt at 1Hz
TCCR1A = 0;// set entire TCCR1A register to 0
TCCR1B = 0;// same for TCCR1B
TCNT1 = 0;//initialize counter value to 0
// set compare match register for 1hz increments
OCR1A = 15624;// = (16 * 106)/(1 * 1024) - 1(must be < 65536)
// turn on CTC mode
TCCR1B |= (1 << WGM12);
// Set CS10 and CS12 bits for 1024 prescaler
TCCR1B |= (1 << CS12) | (1 << CS10);
// enable timer compare interrupt
TIMSK1 |= (1 << OCIE1A);
```

#### 4.2.5 Loop Function

The loop function for this lab simply toggles and LED on each iteration, and updates the timer counter on the LCD every 2000th iteration:

#### Listing 4.6 Loop function

```
void loop() {
    digitalWrite(LED1, LED1State);
    LED1State = 1;
    if(iterCounter>=2000){
        mins=timer/120000; //2kHz * 60 secs/min
        secs=(timer%120000)/2000;
        ms=timer%1000;
        iterCounter=0;
        cd.clear();
        lcd.setCursor(0, 0);
        lcd.print(String(String(mins)+":" +String(secs)+"." +String(ms)));
    }
}
```

Listing 4.6 (cont.)

```
    iterCounter++;  
}
```

#### 4.2.6 Interrupts and Handlers

The students are first instructed to 'attach' an interrupt for button 2 and their "reset" handler. This is done inside the setup() function:

Listing 4.7 Attaching interrupt

```
attachInterrupt(digitalPinToInterrupt(button2), reset, RISING);
```

The "reset" function simply resets the global timer counter variable:

```
void reset(){  
    timer=0;  
    return;  
}
```

The skeleton for the timer handler functions (Interrupt Service Routine, ISR) are given. All that needs to be done for these is to toggle LED2 and update the timer counter by 2 on timer0 interrupt, and toggle LED3 on timer1 interrupt. The reason why we update the timer counter by 2 is because timer0's interrupt is set to trigger 2,000 times per second, and we update and display the time since last reset at the millisecond resolution (Section 4.2.2):

Listing 4.8 ISR routine

```
ISR(TIMER0_COMPA_vect)//timer0 interrupt 2kHz toggles pin 8  
    //generates pulse wave of frequency 2kHz/2 = 1kHz (takes two cycles for full  
    //wave-toggle high then toggle low)  
    if (LED2State){  
        digitalWrite(LED2,HIGH);  
        LED2State = 0;  
    }else{
```

#### Listing 4.8 (cont.)

```
    digitalWrite(LED2,LOW);
    LED2State = 1;
}
timer+=2;

ISR(TIMER1_COMPA_vect)//timer1 interrupt 1Hz toggles pin 13 (LED)
//generates pulse wave of frequency 1Hz/2 = 0.5kHz
//(takes two cycles for full wave-toggle high then toggle low)
if (LED3State){
    digitalWrite(LED3,HIGH);
    LED3State = 0;
}else{
    digitalWrite(LED3,LOW);
    LED3State = 1;
}
}
```

### 4.3 ASSIGNMENT 3: ANALOG TO DIGITAL CONVERTERS AND ANALOG I/O

This lab introduces the concept of a voltage divider (Section 2.4.1) in order to explain how the resistive touchscreen on the Amazing Ball system (and other similar devices) works. As discussed in Chapter 5, due to the circumstances that semester, we did not have access to the Amazing Ball setup, so we replaced that component with a potentiometer. Both components are used in the circuits as variable resistors, and are therefore electronic analogs. We also cover Pulse-Width Modulation and Analog to Digital Converters. These topics are mentioned briefly in the labs, as they are studied in more depth during the lectures.

#### 4.3.1 Learning Objectives

In lab 3, students start by learning the principle of a voltage divider (Section 2.4.2), where the ratio of two resistors connected in series can be used to control the voltage at their midpoint. This allows the students to understand how resistive strips and resistive touchscreens work. Students then learn that an analog signal in an electronic circuit is

measured as a varying resistance, and that analog-to-digital converters create the digital approximations of these values by measuring the voltage at their input and quantizing the value to the nearest bit of available resolution. By this point, students understand that they can control digital values by varying the resistance on their circuits.

In order to go in the opposite direction, students learn that it is possible to create analog signals with a digital pin (limited to fully on or fully off) by the use of pulse-width modulation. Students learn that by varying the width of these pulses (at a fixed frequency), they can create analog signals that average to create a voltage that is a fraction of the supply voltage proportional to the on-time.

#### 4.3.2 Common Errors

Even though the lab requirements are descriptive on this distinction, understanding the difference between the analog and digital representation of the voltage values posed some difficulty to some of the students. With a 10-bit ADC, our digital representations have a range of  $[0, 1024]$ , and the analog voltages present at the pin have to be  $[0.00\text{V}, 5.00\text{V}]$  (ground reference to power supply voltage). Converting between the two also confused some students. The ADC receives the digital values, so in order to find the analog equivalent, they must take the digital value at the pin, divide it by the maximum digital value (1024) to find the percentage of that range which is read at the pin, and multiply it by the maximum voltage available (5V).

#### 4.3.3 Grading Requirements

The requirements for Assignment 3 are to use Arduino's ADC to read the position of the potentiometer using the voltage divider concept. Students then display the digital (0-1024) (20%) and analog (0.00V-5.00V) (20%) on the first line of the LCD, and control the position of a "." on the second line of the LCD, which should mimic the current potentiometer setting (25%). The position of the "." should be updated only when the value is changed (10%). They must then use these values to generate a PWM signal to vary the brightness of an LED, which should also follow the position of the potentiometer knob (25%). This teaches students how to generate an average analog value by changing the duty cycle of a PWM signal. LCD operations are omitted from this section for brevity, since they are covered in Assignment 1 (Section 4.1).

#### 4.3.4 Solution

We create the following global variables to keep track of our voltage values:

Listing 4.9 Global variables

```
unsigned int potValue = 0, oldPotValue = 0;  
float voltage = 0.0;  
int dotPosition = 0;
```

Then in the `loop()` function, we read the voltage on pin A2, where the potentiometer is connected, convert it to the 5V range:

Listing 4.10 Reading A2

```
potValue = analogRead(pot);  
voltage = 5.0 * potValue / 1024;
```

To determine the position of the "." on the LCD, we multiply the voltage we calculated from pin A2 by 16, which is the number of pixels on the LCD, and divide it by 5V:

Listing 4.11 Moving the dot

```
dotPosition = (int)(16*voltage/5);
```

And finally, to illuminate the LED we make a call to `analogWrite()` with the `potValue` divided by 4. This division is done because the ADC is 10-bit (1024), and the `analogWrite()` function takes values from 0 to 255 to control the duty cycle:

Listing 4.12 Illuminate LED

```
analogWrite(LED1, potValue/4);
```

## 4.4 ASSIGNMENT 4: JOYSTICK AND SERVO MOTOR

This assignment attempts to create an analog for Amazing Ball part of the previous lab's implementation where the position of the ball controlled the position of the servo motors that actuated the platform. The variable resistance of the touchscreen is again replaced with the potentiometer analog, and the board itself is replaced with a single servo motor. This gives the students the experience of controlling the position of an actuator using a variable resistance input, much like the previous implementation. However, the Amazing Ball system is a more sophisticated setup and a better platform for teaching control theory, and this hardware should be integrated back into these laboratory assignments when in-person instruction resumes.

### 4.4.1 Learning Objectives

The focus of this lab is on teaching how to control a servo motor using a microcontroller. Servo motors use PWM signals to determine the position of the actuator arm. Having learned the principle of a voltage divider in Lab 3, students should understand that the joystick is equivalent to two voltage dividers, one for each axis. They must then extract those values and convert them in order to generate the correct PWM signal for the servo. The calibration process should teach them that analog components have variability, which must be accounted for in their program. These maximum and minimum values measured during calibration should be used to calculate the position of the rotor, instead of hardcoding the values 0 and 1024.

### 4.4.2 Common Errors

A common error for this lab in previous semesters was for the calibration to skip through all the axis measurements on a single button press, resultant from improper switch debouncing. This semester we were careful with the description and requirements to avoid this issue, but it might be a good learning experience to let students stumble into this problem so they have first-hand experience with the importance of debouncing.

### 4.4.3 Grading Requirements

In this lab, students are asked to begin by calibrating their joystick, which is an operation that is routinely performed in real applications. This involves measuring the maximum and

minimum values of each axis of the joystick, storing those values in provided variables inside the program and use those values to calculate the servo position (20%). They must then display the joystick's X and Y coordinates on the LCD, updated in real time (25%), and use the X axis of the joystick to control the servo motor arm position (25%). Using an interrupt, a press of button 2 should lock the servo motor position and LCD display while disregarding joystick movement (15%), and another press should restore the operation of the program (15%).

#### 4.4.4 Solution

We start by attaching an interrupt on pin 9 to our "lock" function, and then calling our "calibrate" function, both done inside setup():

Listing 4.13 Attaching the lock subroutine

```
attachInterrupt(digitalPinToInterrupt(button2), lock, RISING);  
calibrate();
```

#### 4.4.5 Calibration

The calibration is done by registering the maximum and minimum values for each axis of the joystick. This can be used later to compensate for any variations specific to that unit:

Listing 4.14 Calibration

```
void calibrate(){  
    lcd.setCursor(0, 0);  
    lcd.print("Calibrate X min");  
    waitForPress(button1);  
    _xmin = analogRead(x_in);  
    waitForRelease(button1);  
    lcd.setCursor(0, 0);  
    lcd.print("Calibrate X max");  
    waitForPress(button1);
```

Listing 4.14 (cont.)

```

    _xmax = analogRead(x_in);
    waitForRelease(button1);
    lcd.setCursor(0, 0);
    lcd.print("Calibrate Y min");
    waitForPress(button1);
    _ymin = analogRead(y_in);
    waitForRelease(button1);
    cd.setCursor(0, 0);
    lcd.print("Calibrate Y max");
    waitForPress(button1);
    _ymax = analogRead(y_in);
}

```

During calibration, students are instructed to have their code wait for debounced button presses and releases, so the calibration doesn't run through all the parameters on the first button press. We therefore make use of the debounce function from Assignment 1 (Section 4.1) to write the "waitForPress" and "waitForRelease" functions:

Listing 4.15 Waiting for press and release

```

int waitForPress(int pin){
    (!button1State)
        button1State = debounce(button1);
    }
    return;
}

int waitForRelease(int pin){
    while(button1State)
        button1State = debounce(button1);
    }
    return;
}

```

#### 4.4.6 Locking and Unlocking the Board

To lock the board, we write the "lock" interrupt handler that toggles the "lockedState" global variable, and the entirety of the contents of the loop() function (with the exception of printing the locked/unlocked state) is run conditionally on the value of that variable. This prevents any updates to the motor and LCD, effectively locking the board:

Listing 4.16 The lock function

```
lock()
  unsigned long currentTime = millis();
  if ((currentTime - lastLockDebounce) > debounceDelay) {
    lockedState = !lockedState;
    lastLockDebounce = currentTime;
  }
}
```

#### 4.4.7 Loop

As mentioned above, the contents of the loop() function are run conditionally on the value of the lockedState variable. This results in the desired locked state behavior. The only portion of the code that is not contained in this conditional block is the code that displays the locked/unlocked state of the board on the LCD.

We start by reading the values of A0 and A1 (the joystick's X and Y values respectively) and convert them to the range [-49,+50], as the center position of each axis is reference 0. We then update the minimum and maximum values seen for each axis Only xMin is shown here as xMax, yMin and yMax are nearly identical, but all are included in the solution (Appendix A.4.3). Following this, we control the position of the servo motor with a call to servo.write, containing the position of the servo motor. This position is calculated by taking the analog reading of A0 (joystick X axis valued between 0 and 1024), dividing it by 1024 to find the percentage of the range that value represents, multiply it by 100 to put it into the range [0,100], and finally subtract 49 from it to convert it to the range [-49,50] as required by the assignment:

Listing 4.17 The loop function

```
loop()
{
    if(!lockedState){
        xIn = analogRead(x_in);
        yIn = analogRead(y_in);
        //convert position value
        xVal = (int)((((float)xIn)/1024)*100)-49;
        yVal = (int)((((float)yIn)/1024)*100)-49;
        if(xVal < xMin)
            xMin = xVal;
    }
    //control servo
    servoPos = (int)((((float)xIn)/1024)*180);
    servo.write(servoPos);
}
lcd.setCursor(15, 1);
if(lockedState)
    lcd.print("L");
}else{
    lcd.print("U");
}
}
```

## 4.5 ASSIGNMENT 5: SERIAL COMMUNICATION

Up to this point, all computer-to-Arduino communications have relied on the Arduino IDE and its built-in facilities. In this lab, students learn to transmit information directly between the Arduino and a Python program running on a PC. This greatly expands the types of projects that can be created, offloading heavy computation to a desktop computer or a server, while communicating actuator and sensor information to and from the microcontroller unit. This principle can also be used to communicate between microcontroller boards.

This lab uses the Python Serial Library (pyserial) for serial communications, and optional GUI using Tkinter. This latter part is optional since some students may have difficulty

installing the package. A terminal version of the same program is provided as an alternative. As an illustration of the applications of these principles, this assignment is a simplified derivative of a 2-channel oscilloscope project I have designed in the past.

#### 4.5.1 Learning Objectives

In this lab, students learn how to use serial communication to send from their board to other devices. They learn how to find and address their device on their own desktop Python programs, and how to use the pyserial and Arduino's Serial libraries to do so. This lab also exposes them to a Python library for generating Graphical User Interfaces, called TKInter. This part of the lab is optional, and a terminal-only version of the assignment is provided in the code, but being familiar with how to create GUI's in Python is a useful skill for all programmers.

#### 4.5.2 Common Errors

A common error students should look out for is using the correct baud rate. The baud rate used in the initialization of the serial communication in the Arduino (`Serial.begin(115200)`) needs to match the baud rate being received by the pyserial library (`ser = serial.Serial('/dev/cu.usbmodem143201',115200, timeout=1)`)

#### 4.5.3 Grading Requirements

The requirements for this assignment are to display a prompt on the LCD screen connected to the Arduino (10%), then write a message on the Python GUI on the desktop computer and send it to the Arduino (20%). The Arduino must display the message that was sent (in this case, the student's name) (20%). After this, the Arduino should continually send the analog readings from the pin connected to the potentiometer (pin A2) to the Python program, which then displays the voltage on the computer screen, updated in real time (50%). This should be done in the `loop()` function, by converting A2's 10-bit value to the [0v, 5V] range, as done in previous assignments, and sent to the computer with a simple `serialWrite` call, also seen in previous labs.

#### 4.5.4 Read String Function

The students must write a `readString()` function which is called by `setup()`. This function

should wait until serial data is available from the computer, checking every 50ms, then once data is available, it should read the data character by character into the global myName[] buffer, until there is no more available data or if MAX\_STR value characters have been read. The function should then null-terminate the string:

Listing 4.18 Readstr function

```
void readStr(char *str){
    char c;
    int i=0;
    while(!Serial.available()){
        delay(50);
    }
    while(Serial.available() && i<MAX_STR){
        str[i]=Serial.read();
        i++;
        delay(2); //delay to allow byte to arrive in input buffer
    }
    str[-i]="";
    return;
}
```

#### 4.5.5 PC Python Program

The python code is entirely provided in the starter code. The only necessary change the students must make is to find the address of the board on their computers in the Arduino IDE, and update the first argument of the serial.Serial initializing function at the top of the Python script to reflect it. This address can be found in the Arduino IDE, or through inspecting the contents of the /dev folder in a Unix terminal.

Once the board is connected and turned on, and the Arduino code has been completed, the Python script can be run.

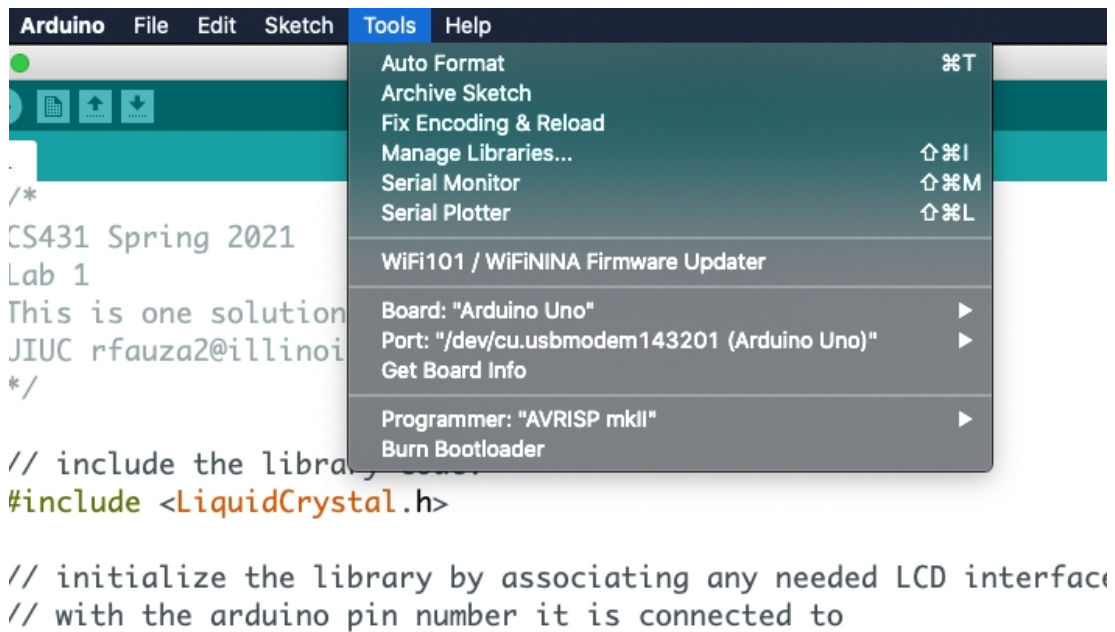


Figure 4.3: Finding Arduino Serial Address in IDE

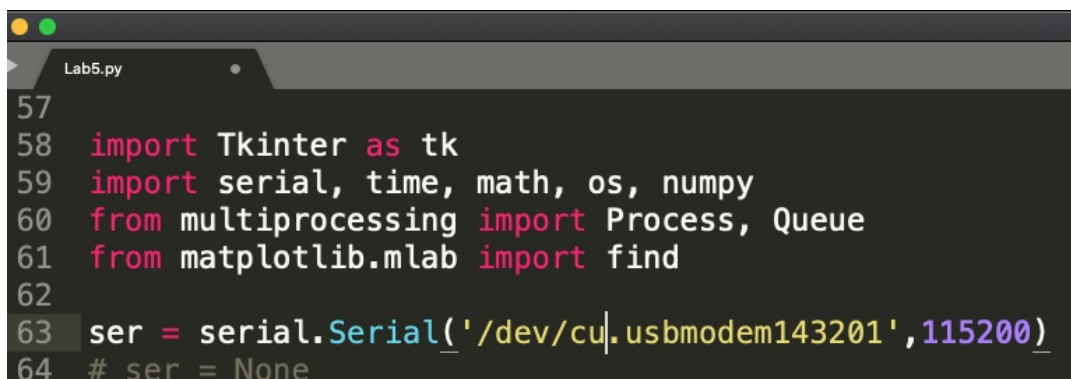


Figure 4.4: Updating Arduino Serial Address on Python Program

## CHAPTER 5: FUTURE WORK

For future semesters, the complexity of the labs should be increased, as they will be completed in-person with T.A. present while students work on their projects. The Arduino platform provides abstractions of low-level programming that are great for learning all the concepts taught in our class and provides easier access to first-hand experience of the theoretical material. But introducing some direct programming of the ATmega328P on labs later in the semester would be a valuable experience for the students. Similarly, given code in future semesters can be less complete, and more advanced requirements may be added. With shared university hardware available, it would also be important to show students how to generalize what they learned in this course to other embedded platforms, including the RaspberryPi and classic standalone microcontrollers like the dsPIC33 used in previous semesters.

I am of the strong opinion that each student building their kit and learning about the basic electronics principles behind it is an important component of Embedded Programming that our future labs should retain. And finally, the resistive touchscreen and balancing ball from the Amazing Ball hardware should be reintegrated into the labs for future semesters using our Arduino Uno boards. The current kits we had available this semester unfortunately provided no analog for a proper project on control theory.

## CHAPTER 6: CONCLUSIONS

The laboratory component of our class is very important to the learning and retention of the subject of Embedded Systems. A class covering this material without the hands-on experience would be of very limited value to someone who intends to use this knowledge in the field, and the experience of learning to interface your software with physical systems is very rewarding and enables the individual to explore a whole new class of projects not possible without it.

Due to COVID-19, all instruction was done remotely for the Spring semester of 2021. This imposed a particular difficulty in the teaching of our class. The laboratory assignments require a physical setup, while most students who sign up for the class do not have any background in electronics, as this is not covered in the normal Computer Science curriculum. To mitigate this issue, the first lab was designed to cover, in detail, the step-by-step process of setting up the board for all our labs, and ample time was available for one-on-one instruction and troubleshooting with the T.A.

A few of the students joined the class with previous experience and had no problems, while others required some help to set up their boards correctly. In the end, all students had their lab setups working properly within reasonable time. In previous semesters, the boards had already been setup before the first lab, which simplified the process, bypassed students having to troubleshoot without experience, and removed the requirement for the students to acquire some of the basics electronics knowledge used in our class of Spring 2021.

It is my firm opinion, however, that each student setting up their own board, learning about the basic electronics principles surrounding it, troubleshooting and watching their circuits come alive provided the students with valuable additional knowledge that they would not have gained otherwise. For that reason, I think this part of the process should be included in future classes. The fact that our more modern platform allows less time to be spent with the intricacies of low-level programming of the microcontroller chip and its registers, as opposed to the previous platform used (AVR), allows for this time to be spent learning this broader breadth of useful knowledge.

Some confusion in the labs stemmed from small revisions having to be made to some assignments due to pinout compatibility: I originally tried to simplify the wiring by using sequentially numbered pins for the connections to the LCD on the first lab (pins 3, 4, 5 and

6), but when creating Lab 3, I had to switch the LED from pin 7 to pin 6, since pin 7 is not capable of generating a PWM signal. This problem could have been easily addressed during off-hours had I had access to the devices before the next lab.

After the boards were setup, few students made use of office hours for help with their lab assignments, which I believe reflects that the labs were reasonable to follow and execute by the students independently. In some cases, there were students who misunderstood some of the requirements of the labs, and had these labs been presented in person as they were when I was a student in this class, the T.A. would have been able to point out the errors and let the students correct them and present again within the allotted lab hours.

It was also difficult for me as an instructor to gauge the difficulty the students were having with the assignments. My intention was to give them more advanced knowledge with less descriptive guidance, but I noticed that some of the students in the class were too new to the subject and weren't ready for so much autonomy. As a result, I made the labs more general and easier to follow, hoping to cover a broader range of student ability in the subject. It was my reservation, then, that the more advanced students who hoped to gain more advanced knowledge may have found the class not challenging enough. With a T.A. physically available while the students complete their assignments during the scheduled lab times, more demanding requirements may be introduced into the labs for future semesters. Extra credit sections with more in-depth and low-level control of the machine may also be added for those students who want more, though those students might not need the extra grade points.

In conclusion, my experience has been that delivering this class remotely provided a few new challenges, but these challenges did not deter the successful delivery of the class material, knowledge and experience in the subjects covered in the lecture or labs.

## REFERENCES

- [1] C. Platt, *Make: Electronics*. Sebastopol, CA: O’Riley, 1972.
- [2] “Erika enterprise flex boards,” <http://www.erika.tuxfamily.org/drupal/flex-board.html>  
note = Accessed: 2021-10-2.
- [3] “Microchip dspic33f family datasheet,” <http://ww1.microchip.com/downloads/en/DeviceDoc/70165d.pdf>, accessed: 2021-10-2.
- [4] “Erika enterprise amazing ball,” <https://erika.tuxfamily.org/drupal/amazing-ball.html>  
note = Accessed: 2021-10-2.
- [5] “Arduino uno,” <https://store-usa.arduino.cc/products/arduino-uno-rev3>, accessed: 2021-10-2.
- [6] “Atmega328p datasheet,” [http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P\\_Datasheet.pdf](http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf), accessed: 2021-10-2.
- [7] “Arduino software,” <https://www.arduino.cc/en/software>, accessed: 2021-10-2.
- [8] “Arduino reference,” <https://www.arduino.cc/reference/en/>, accessed: 2021-10-2.
- [9] “Arduino introduction,” <https://www.arduino.cc/en/Guide/Introduction>, accessed: 2021-10-2.
- [10] “Arduino port manipulation,” <https://www.arduino.cc/en/Reference/PortManipulation>, accessed: 2021-10-2.
- [11] “Arduino serial communication,” <https://www.arduino.cc/reference/en/language/functions/communication/serial/>, accessed: 2021-10-2.
- [12] “Arduino blink,” <https://www.arduino.cc/en/Tutorial/BuiltInExamples/Blink>, accessed: 2021-10-2.
- [13] “Arduino libraries,” <https://www.arduino.cc/reference/en/libraries/>, accessed: 2021-10-2.
- [14] “Arduino liquid crystal library,” <https://www.arduino.cc/en/Reference/LiquidCrystal>, accessed: 2021-10-2.
- [15] “Arduino servo library,” <https://www.arduino.cc/en/Reference/servo>, accessed: 2021-10-2.

## APPENDIX A: LABORATORY ASSIGNMENTS

This Appendix presents the instruction slides and unabridged starter and solution code for each assignment given this semester.

## A.1 ASSIGNMENT 1 SETUP AND DIGITAL I/O

### A.1.1 Instruction Slides

# Lab 1

## Setup and Digital I/O

Figure A.1: Lab 1 Slide 1

### Arduino Kit

- Email the TA your shipping address ASAP so you don't miss the first deadline.
- We will mail you a kit like this
- Students outside the U.S. might need special arrangements with CS department
- Return kit at end of semester: Keep all boxes and papers, all parts in working order
- You will be responsible for any broken or missing parts



Figure A.2: Lab 1 Slide 2

## Multimeter

The multimeter is an essential tool for working with electronic circuits. On its screen, it shows the voltage difference between its two probes. It can also be used to measure current, resistance, and even temperature. The black probe should be connected to the COM (common) insert, and is usually connected to ground. The red probe should be connected to the V insert (on the right in this picture), and is used to probe the circuit. The 10A insert is used for high current measurements and should be left disconnected for our class.

The knob in the middle selects the type and range of measurement we intend to make: The markings on the upper left of the knob are for making direct current measurements. The two on the upper right are for alternating currents. And the ones on the bottom left are used for measuring resistance values.

Our Arduino board runs on 5V DC, so you should set the knob to 20V DC before taking measurements.



Figure A.3: Lab 1 Slide 3

## Oscilloscope

The oscilloscope is a very powerful electronics measurement equipment that gives insight into the inner workings of our circuits. Much like the multimeter, it measures voltages, but it plots these on the screen in real time, allowing us to see the signals as they flow through our circuit.

We can adjust the vertical and horizontal scales of the displayed signals with the two corresponding knobs on each channel. The alligator clip on the test lead is connected to the circuit ground, and the test tip can be used to probe the circuit. The oscilloscope usually plots signals continuously. But it can also be used to catch very fast signal events so we can study them in more detail.



Figure A.4: Lab 1 Slide 4

## Arduino Programming

- Install Arduino IDE on your computer (<https://www.arduino.cc/en/software>)
- Open the IDE and connect the Arduino via USB
- Select correct board (Uno) and port under the “tools” menu
- Load built-in “Examples/Basics/Blink” sketch for testing (recommended)
- Click “upload” button in the upper left corner of IDE. The built-in LED should start blinking once per second after upload is complete
- Arduino Pins: <https://www.arduino.cc/en/reference/board>
- Connecting anything to digital pins 0 and 1 will prevent the board from being programmed, so leave those pins disconnected

Figure A.5: Lab 1 Slide 5

## Arduino Programming

Microcontrollers run in an infinite loop. You will have a `setup()` function and a `loop()` function which the Arduino knows how to call (see built-in examples in the IDE and also on [www.arduino.cc](http://www.arduino.cc)). You may create other functions and call them from `setup()` or `loop()`, but these two functions are required

Arduino is programmed in a variant of C.

- Include `LiquidCrystal.h` library header
- Call `lcd.begin(16, 2)` in your setup loop
- Set LED pins to output mode, and switch pins to input mode

Figure A.6: Lab 1 Slide 6

## Arduino Programming: Arduino C

```
#include <LiquidCrystal.h>
// includes and variable declarations
const int LED1 = 6, button1 = 10;

void setup() {
  // set up the LCD's number of columns and rows:
  lcd.begin(16, 2);
  pinMode(LED1, OUTPUT);
  pinMode(button1, INPUT);
  // put your setup code here, to run once:
}

void loop() {
  // put your main code here, to run repeatedly:
  delay(100);
}
```

Figure A.7: Lab 1 Slide 7

## Breadboard

- All pins on each numbered row (for each half of the board) are connected together. So A-E for each row number are all connected internally (same for F-J). Each leg of a component should go on a different numbered row

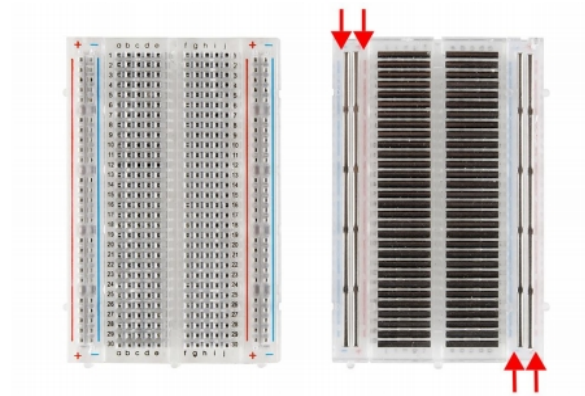


Figure A.8: Lab 1 Slide 8

## Breadboard

- The + and - rails on either side of the board are often disconnected at the center. You might have to bridge these connections to use the rails across the entire board
- Connect the + rail to the 5v pin of the Arduino, and the - rail to the GND pin of Arduino

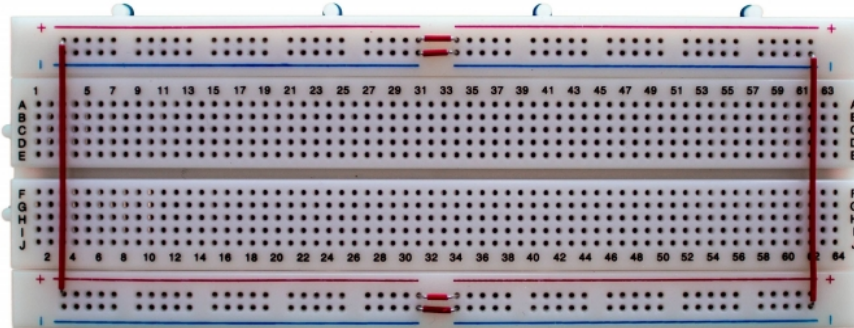


Figure A.9: Lab 1 Slide 9

## Resistors

- Resistance is denoted by color bands on the resistor, as in the picture
- First 3 bands are digits (first 2 on resistors with only 4 bands), 4th band is power of 10, 5th band is tolerance (ignore)
- E.g. 560k Ohm:
  - Band 1: Green = 5
  - Band 2: Blue = 6
  - Band 3: Black = 0
  - Band 4: Orange = 3
  - $560 * 10^3 = 560,000$

Find Tolerance Band (Usually Separated) and work from other side

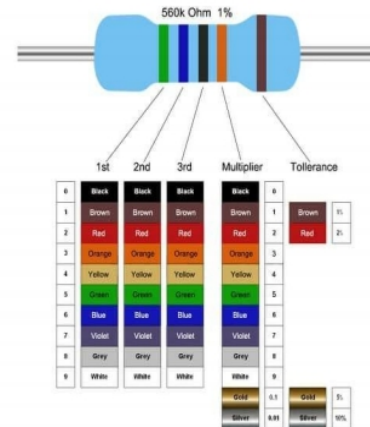


Figure A.10: Lab 1 Slide 10

## LED

- DO NOT connect LED's directly between power and ground. LED's require a resistor or they will burn out immediately
- Resistor can be on positive or negative side (doesn't matter)
- Standard resistor values for LED's are 330 Ohm and 470 Ohm
- Higher resistances will work but light will be dimmer

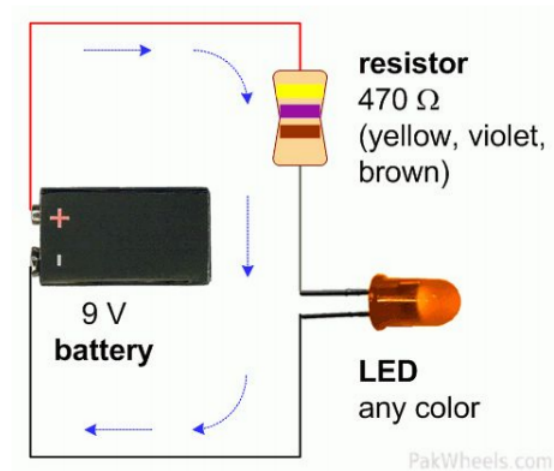


Figure A.11: Lab 1 Slide 11

## LED

- Longer leg connects to Positive
- Shorter leg connects to Negative
- IMPORTANT: Do not connect it in reverse
- Do not clip or cut the legs
- Connect the Arduino digital pin to the Positive leg of the LED, the negative leg to a resistor, and the other side of the resistor to ground
- Connect LEDs to pins 13, 7 and 8

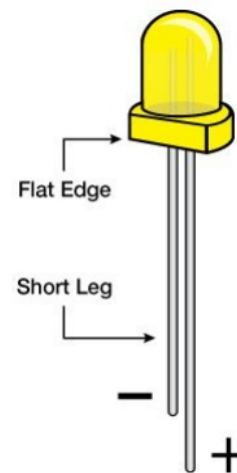


Figure A.12: Lab 1 Slide 12

### A.1.2 Given Code

Listing A.1 Lab 1 given code

```
/*
CS431 Spring 2021
Lab 1 - Digital IO
Display your name on the screen on the first line of the LCD
Display a count of debounced switch 1 presses on the second line
LED 1 (pin 13) turns on when switch 1 (pin 10) is on
LED 2 (pin 7) turns on when switch 2 (pin 2) is on
LED 3 (pin 8) turns on when switch 1 and 2 are in the same state (both pressed or both
released)
Submit to Compass: your code (lab1-netid.ino) and a quick video (lab1-netid.mp4 or other
format) showing the board behaving as expected

UIUC rfauza2@illinois.edu
*/

// include the library code:
include <LiquidCrystal.h>

// initialize the library by associating any needed LCD interface pin
// with the Arduino pin number it is connected to
// TO DO: double-check your connections to make sure all the pins are connected as num-
bered below
const int rs = 12, en = 11, d4 = 3, d5 = 4, d6 = 5, d7 = 6;
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);
const int button1 = 10, button2 = 2, LED1 = 13, LED2 = 7, LED3 = 8;
unsigned long debounceDelay = 50;
// the setup function runs once when you press reset or power up the board
void setup() {
//TO DO:
//enter your setup code here
}
```

### Listing A.1 (cont.)

```
// the loop function runs over and over again forever
void loop() {
  //TO DO:
    //enter your loop code here
}
  int debounce(int pin){
// You may want to write your debunce in a separate function (optional)
int state = 0;
return state;
}
```

#### A.1.3 Solution Code

### Listing A.2 Lab 1 solution code

```
/*
CS431 Spring 2021
Lab 1 - Digital IO
Display your name on the screen on the first line of the LCD
Display a count of debounced switch 1 presses on the second line
LED 1 (pin 13) turns on when switch 1 (pin 10) is on
LED 2 (pin 7) turns on when switch 2 (pin 2) is on
LED 3 (pin 8) turns on when switch 1 and 2 are in the same state (both pressed or both
released)
Submit to Compass: your code (lab1-netid.ino) and a quick video (lab1-netid.mp4 or other
format) showing the board behaving as expected

UIUC rfauza2@illinois.edu
*/
// include the library code:
include <LiquidCrystal.h>
// initialize the library by associating any needed LCD interface pin
// with the Arduino pin number it is connected to
const int rs = 12, en = 11, d4 = 3, d5 = 4, d6 = 5, d7 = 6;
```

## Listing A.2 (cont.)

```

LiquidCrystal lcd(rs, en, d4, d5, d6, d7);
unsigned long debounceDelay = 50;
const int button1 = 10, button2 = 2, LED1 = 13, LED2 = 7, LED3 = 8;
    int button1State = 0;
int button2State = 0;
unsigned long button1Presses = 0;
int prevButtonStates[2] = 0,0;
unsigned long lastDebounce[2] = 0, 0;
    // the setup function runs once when you press reset or power the board
void setup() {
    // allows print statements to console (Tools->Serial Monitor)
    Serial.begin(9600);
    // initialize digital pin LED_BUILTIN as an output.
    pinMode(button1, INPUT);
    pinMode(button2, INPUT);
    pinMode(LED1, OUTPUT);
    pinMode(LED2, OUTPUT);
    pinMode(LED3, OUTPUT);
    // set up the LCD's number of columns and rows:
    lcd.begin(16, 2);
    // Print a message to the LCD.
    lcd.setCursor(0, 0);
    lcd.print("Welcome to CS431");
    lcd.setCursor(0, 1); // set the cursor to column 0, line 1
    lcd.print(0);
}
    // the loop function runs over and over again forever
void loop() {
    //Debounce Buttons
    button1State = debounce(button1);
    button2State = debounce(button2);
    Serial.println("B1:" + String(button1State) + " Prev:" + String(prevButtonStates[0]));
    if(button1State != prevButtonStates[0]){
        digitalWrite(LED1, HIGH);
    }
}

```

Listing A.2 (cont.)

```

prevButtonStates[0] = 1;
button1Presses++;
    lcd.setCursor(0, 1);
    lcd.print(button1Presses);
}
if(!button1State  prevButtonStates[0])
prevButtonStates[0] = 0;
digitalWrite(LED1, LOW);
}
if(button2State  !prevButtonStates[1]){
prevButtonStates[1] = 1;
digitalWrite(LED2, HIGH);
}
if(!button2State  prevButtonStates[1]){
prevButtonStates[1] = 0;
digitalWrite(LED2, LOW);
}
if(button1State==button2State){
digitalWrite(LED3, HIGH);
}else{
digitalWrite(LED3, LOW);
}
}
    int debounce(int pin){
int state = 0;
if(pin != 10  pin != 2){
return 0;
}
int button=0;
if(pin==2){
button = 1;

    unsigned long currentTime = millis();
if ((currentTime - lastDebounce[button]) > debounceDelay) {

```

Listing A.2 (cont.)

```
state = digitalRead(pin);  
lastDebounce[button] = currentTime;  
}else{  
    state = prevButtonStates[button];  
}  
return state;  
}
```

## A.2 ASSIGNMENT 2 TIMER REGISTERS AND INTERRUPTS

### A.2.1 Instruction Slides

# Lab 2

## Timer Registers and Interrupts

Figure A.13: Lab 2 Slide 1

### Registers

Registers are protected bytes in the computer's memory space which are generally used internally by the system, but can be accessed by programmers. For example, in the Arduino, pins 0-7 direction (set to input or output) are controlled by the register named DDRD (predefined macro), so to directly set the state of these pins, you can set DDRD to the desired mask value (e.g. `DDRD = b01100100;`)

Read the Arduino documentation on registers:

<https://www.arduino.cc/en/Reference/PortManipulation>

Figure A.14: Lab 2 Slide 2

## Arduino Timers and registers

Timers are counters that are incremented at predefined multiples of the CPU clock. The Arduino has 3 built-in timer registers: Timers 0 and 2 are 8-bit timers ( $2^8$  time subdivisions), and Timer 1 is 16-bit ( $2^{16}$  time subdivisions), so Timer 1 allows for better precision. Arduino registers are usually hidden from the end user for ease of programming, but they are used by libraries (e.g. Timer 0 is used by built-in functions like `delay()` and `millis()`). As an Embedded System Programmer, you will be required to know how to work with registers directly.

The following website provides a very good explanation with examples:

<https://www.instructables.com/Arduino-Timer-Interrupts/>

Figure A.15: Lab 2 Slide 3

## Prescaler

An important value to understand is the prescaler. This is essentially the multiple of the clock frequency at which a timer register will be incremented.

Desired Frequency (Hz) = Clock Speed (16MHz) / prescaler

For example, if you want a 2 MHz frequency on the Arduino, you need a 8 prescaler

Figure A.16: Lab 2 Slide 4

## Polling

Microcontrollers run the code in the `loop()` function in an infinite loop, but sometimes we want the behavior to change based on some parameter change (e.g. the press of a button, or the temperature in an engine getting too high). One way to achieve this is to check the value of a pin on every loop and call a function if the pin is in one state (e.g. high), and another (or just skip it) if the state is the opposite. This is called “Polling”.

Figure A.17: Lab 2 Slide 5

## Interrupts

Polling every pin and register on every loop would be very inefficient and could delay response, so microcontrollers often provide interrupts. An interrupt is an internal flag (a physical bit that is turned high or low) that indicates that an external event has occurred. Once this flag is set, the microcontroller will immediately stop the execution of the loop and “handle” the interrupt (run a predefined function for that particular interrupt). This is similar in behavior to interrupts used in C programming. The “external” event that triggers an interrupt can also be the value of an internal register.

Read the Arduino reference material on pin interrupts:

<https://www.arduino.cc/reference/en/language/functions/external-interrupts/attachinterrupt/>

Optional: C interrupts for reference:

<http://www.signal.uu.se/Staff/pd/DSP/Doc/ctools/apxc.pdf>

Figure A.18: Lab 2 Slide 6

## Arduino Pin Interrupts

The Arduino Uno board can only use pins 2 and 3 for interrupts. The starter code already uses the correct numbers, so make sure your connections follow the directions:

- Switch 2 on pin 2
- LCD D4 to pin 3
- LCD D4 to pin 4
- LCD D4 to pin 5
- LCD D4 to pin 6

Figure A.19: Lab 2 Slide 7

## Lab 2 Requirements

- Toggle LED1 inside your main program loop on every iteration.
- Toggle LED2 every 0.5 milliseconds using Timer 0's interrupt.
- Toggle LED3 every 1 second using Timer 1's interrupt.
- If your LEDs are dim for this project, don't worry, as long as we can see them blinking in the video
- In the main loop, display on the first line the time since the last reset (update every 2000th iteration).  
The format should be mm:ss.xxx where mm is minutes, ss is seconds, and xxx is milliseconds.
- Reset the time to 00:00.000 when Button 2 is pressed using rising edge-triggered external interrupt (not by polling the button). Read the documents about interrupts to see how to properly enable the interrupt.

Figure A.20: Lab 2 Slide 8

### A.2.2 Given Code

Listing A.3 Lab 2 given code

```
/*
CS431 Spring 2021
Lab 2 - Interrupts
This is one solution, your code may be different
UIUC rfauza2@illinois.edu
*/
// include the library code:
#include <LiquidCrystal.h>

// initialize the library by associating any needed LCD interface pin
// with the Arduino pin number it is connected to
const int rs = 12, en = 11, d4 = 3, d5 = 4, d6 = 5, d7 = 6;
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);
unsigned long debounceDelay = 100;
const int button1 = 10, button2 = 2, LED1 = 13, LED2 = 7, LED3 = 8;
const int x_in = A0;
const int y_in = A1;
int button1State = 0;
int prevButtonStates[2] = {0,0};
unsigned long lastDebounce[2] = {0, 0};
    unsigned int mins=0, secs=0, ms=0, iterCounter=0,
resetCounter=0;
int LED1State=0, LED2State=0, LED3State=0;
    unsigned long int timer=0;
// the setup function runs once when you press reset or power the board
void setup()
Serial.begin(9600);
// TO DO: initialize LEDs as an output.
// set up the LCD's number of columns and rows:
lcd.begin(16, 2);
cli();//stop interrupts
//set timer0 interrupt at 2kHz (1sec / 2kHz = 0.5ms)
```

### Listing A.3 (cont.)

```

    TCCR0A = 0; // set entire TCCR0A register to 0
    TCCR0B = 0; // same for TCCR0B
    TCNT0 = 0; // initialize counter value to 0
    // set compare match register for 2khz increments
    OCR0A = 124; // = (16*10^6)/(2000 * 64) - 1 (must be < 256)
    //turn on CTC mode
    TCCR0A |= (1 << WGM01);
    //Set CS01 and CS00 bits for 64 prescaler
    TCCR0B |= (1 << CS01) | (1 << CS00);
    //enable timer compare interrupt
    TIMSK0 |= (1 << OCIE0A);
    //TO DO:
    //set timer1 interrupt at 1Hz
    //...
    sei(); //allow interrupts
    //To do:
    //use the attachInterrupt() to bind the reset function on the rising edge of button2
    attachInterrupt(/* ... */);
}
// the loop function runs over and over again forever
void loop()
    //TO DO:
    //Your code here
}
    void reset()
    //TO DO:
    //Your code here
}
//Interrupt Handlers:
ISR(TIMER0_COMPA_vect) //timer0 interrupt 2kHz toggles pin 8
//generates pulse wave of frequency 2kHz/2 = 1kHz (takes two cycles for full wave-toggle
high then toggle low)
//TO DO:
    //Toggle LED2, update timer

```

### Listing A.3 (cont.)

```
}
  ISR(TIMER1_COMPA_vect)//timer1 interrupt 1Hz toggles pin 13 (LED)
//generates pulse wave of frequency 1Hz/2 = 0.5kHz (takes
two cycles for full wave-toggle high then toggle low)
  //TO DO:
//Toggle LED3
}
```

### A.2.3 Solution Code

#### Listing A.4 Lab 2 solution code

```
/*
CS431 Spring 2021
Lab 2 - Interrupts
This is one solution, your code may be different
UIUC rfauza2@illinois.edu
*/

// include the library code:
#include <LiquidCrystal.h>

// initialize the library by associating any needed LCD interface pin
// with the arduino pin number it is connected to const int rs = 12, en = 11, d4 = 3, d5 =
4, d6 = 5, d7 = 6;
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);
unsigned long debounceDelay = 100;
const int button1 = 10, button2 = 2, LED1 = 13, LED2 = 7, LED3 = 8;
  const int x_in = A0;
const int y_in = A1;
int button2State = 0;
  int prevButtonStates[2] = 0,0;
  unsigned long lastDebounce[2] = 0, 0;
```

Listing A.4 (cont.)

```

unsigned int mins=0, secs=0, ms=0, iterCounter=0,
resetCounter=0;
int LED1State=0, LED2State=0, LED3State=0;
    unsigned long int timer=0;
    // the setup function runs once when you press reset or power the board
void setup() {
Serial.begin(9600);
// initialize digital pin LED_BUILTIN as an output.
pinMode(LED1, OUTPUT);
pinMode(button1, INPUT);
pinMode(button2, INPUT);
// set up the LCD's number of columns and rows:
lcd.begin(16, 2);
cli();//stop interrupts

//set timer0 interrupt at 2kHz (1sec / 2kHz = 0.5ms)
TCCR0A = 0;// set entire TCCR0A register to 0
TCCR0B = 0;// same for TCCR0B
TCNT0 = 0;//initialize counter value to 0
// set compare match register for 2khz increments
OCR0A = 124;// = (16 * 106)/(2000 * 64) - 1 (must be <256)
// turn on CTC mode
TCCR0A —= (1 << WGM01);
// Set CS01 and CS00 bits for 64 prescaler
TCCR0B —= (1 << CS01) — (1 << CS00);
// enable timer compare interrupt
TIMSK0 —= (1 <_ OCIE0A);

//set timer1 interrupt at 1Hz
TCCR1A = 0;// set entire TCCR1A register to 0
TCCR1B = 0;// same for TCCR1B
TCNT1 = 0;//initialize counter value to 0
// set compare match register for 1hz increments
OCR1A = 15624;// = (16*106)/(1 * 1024) - 1

```

# Listing A.4 (cont.)

```

(must be <65536)
// turn on CTC mode
TCCR1B —= (1 << WGM12);
// Set CS10 and CS12 bits for 1024 prescaler
TCCR1B —= (1 << CS12) | (1 ÿ CS10);
// enable timer compare interrupt
TIMSK1 —= (1 << OCIE1A);

sei();//allow interrupts
  attachInterrupt(digitalPinToInterrupt(button2), reset, RISING);
}
  // the loop function runs over and over again forever
void loop() {
  digitalWrite(LED1, LED1State);
  LED1State ^ = 1;
  if(iterCounter >=2000){
    mins=timer/120000; //2kHz * 60 secs/min
    secs=(timer%120000)/2000;
    ms=timer%1000;
    iterCounter=0;
    lcd.clear();
    lcd.setCursor(0, 0);
    lcd.print(String(String(mins)+":" +String(secs)+"." +String(ms)));
  }
  iterCounter++;

  void reset(){
    timer=0;
    return;
  }
  ISR(TIMER0_COMPA_vect)//timer0 interrupt 2kHz toggles pin 8
  //generates pulse wave of frequency 2kHz/2 = 1kHz (takes two cycles for full wave-toggle
  high then toggle low)
    if (LED2State)

```

Listing A.4 (cont.)

```
digitalWrite(LED2,HIGH);
    LED2State = 0;
}
}else{
digitalWrite(LED2,LOW);
LED2State = 1;

timer+=2;
}
ISR(TIMER1_COMPA_vect)//timer1 interrupt 1Hz toggles pin 13 (LED)
//generates pulse wave of frequency 1Hz/2 = 0.5kHz (takes two cycles for full wave-toggle
high then toggle low)
if (LED3State){
digitalWrite(LED3,HIGH);
LED3State = 0;
} else
{
digitalWrite(LED3,LOW);
LED3State = 1;
}
}
```

## A.3 ASSIGNMENT 3 ANALOG TO DIGITAL CONVERTERS AND ANALOG I/O

### A.3.1 Instruction Slides

# Lab 3

## ADC, Analog I/O

Figure A.21: Lab 3 Slide 1

### Voltage Dividers

A standard electronics concept is the voltage divider. We can create any output voltage within the range of our power supply by using two resistors. The ratio of these resistors determines the percentage of the supply voltage ( $V_{in}$ ) seen by the output ( $V_{out}$ ). In the picture to the right,  $V_{out}$  will be  $V_{in} * R_2 / (R_1 + R_2)$ . If  $R_1 = R_2$ ,  $V_{out}$  will be exactly 50% of  $V_{in}$ , whether they are both 1 Ohm or 500M Ohm.

As we change this ratio, we change the voltage seen at the output. In this lab,  $R_1$  will be equivalent to the right leg of the potentiometer,  $R_2$  will be the left leg, and  $V_{out}$  will be the center leg. If you followed Lab 1, you should already have these connections.

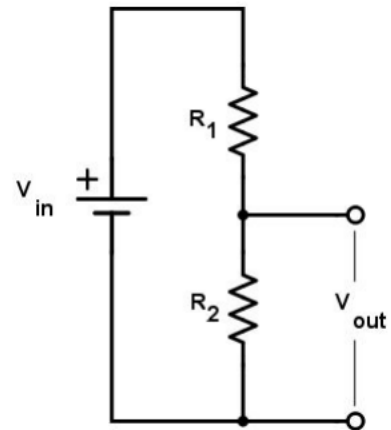


Figure A.22: Lab 3 Slide 2

## Resistive Touchscreen

In previous labs, we used a resistive touchscreen to track the location of and balance a metal ball. The board is made of a series of resistive strips, and it creates a voltage divider (one for each axis) by creating a connection along the strip where its surface is pressed, thereby completing the circuit with the proportional amount of resistance (which as we saw, varies the voltage). The voltage is then read and converted to a digital value by an ADC.

This year, due to work being remote, we will not have access to the original setup. Instead, we will understand the same principles by using the potentiometer as an analog for the variable resistance produced by the touchscreen.

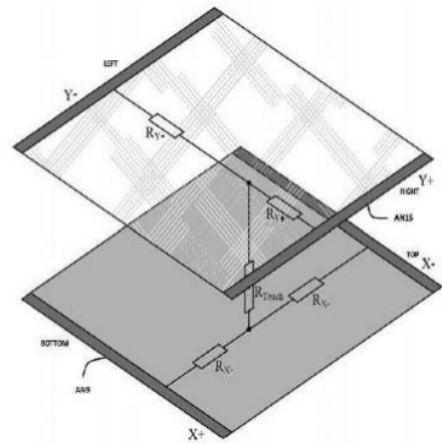


Figure A.23: Lab 3 Slide 3

## Resistive Strip Example

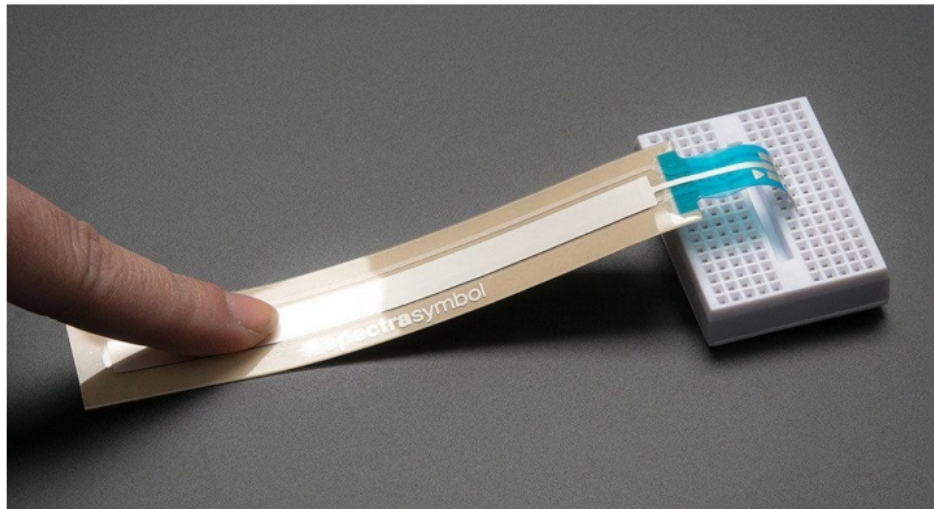


Figure A.24: Lab 3 Slide 4

## Previous Lab Setup

The board is a resistive touchscreen to track ball position

Our lab this year will be different

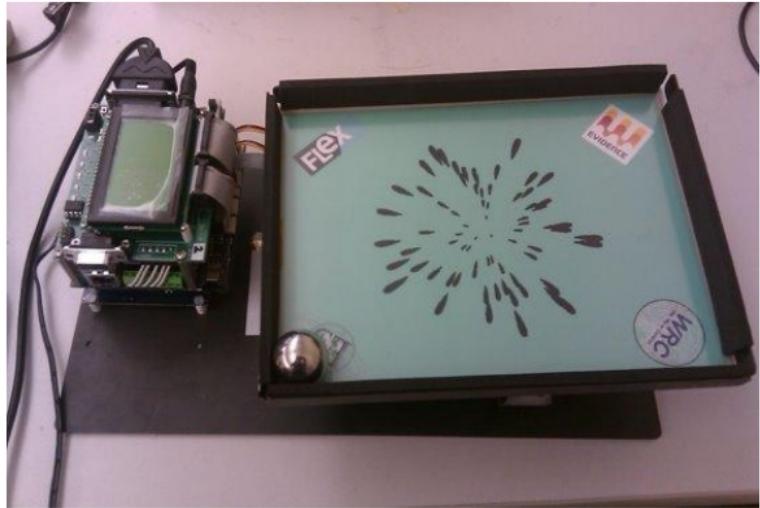


Figure A.25: Lab 3 Slide 5

## Analog to Digital Converters

- Pins A0-A5 on Arduino have 10-bit digital converters which turn analog voltages into discrete digital values
- Joystick pins VRX and VRY, as well as the middle leg of our potentiometer will give us the analog signals we will use
- Your board should already be properly connected from Lab1

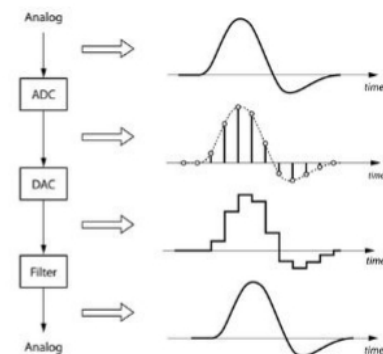


Figure A.26: Lab 3 Slide 6

## Pulse Width Modulation

- PWM works by varying the duty cycle of a digital signal (percentage of time a square wave is high) to vary the average output voltage. This way we can emulate an analog output with a digital pin
- Note that the frequency of the wave remains constant
- Arduino pins marked with a “~” (3,5,6,9,10,11) have this capability
- On the top picture, note that in our setup the lowest voltage ( $-V_a$ ) is equal to 0 (ground), and the reference line in the center is  $\frac{1}{2} V_d = 5v * .5 = 2.5v$

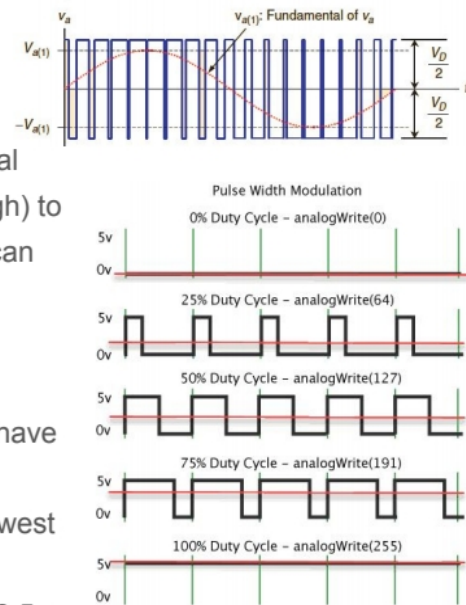


Figure A.27: Lab 3 Slide 7

## Lab 3 Requirements

- Display the digital (0-1024) and analog (0v-5v with 2 decimal digits) values read in by the potentiometer (pin A2) on the the first line of the LCD, updated in real time and labeled (“D:” and “A:” respectively, you will need to convert in your code)
- On the second line, display a ‘.’ that moves left and right as the potentiometer changes. It should correctly show the position of the knob (‘.’ all the way left for knob all the way left, etc)
- Because LCD operations are very slow compared to other processes, update the screen only when the value changes
- Send the PWM signal to LED 1 (pin 6) and control the brightness of the LED by turning the potentiometer (use pin 7 for d7 of the LCD). Your LED should go from off to full brightness.
- Observe the PWM signal with the oscilloscope as you change the potentiometer from minimum to maximum
- Submit to Compass: your code (lab3-netid.ino) and a quick video (lab3-netid.mp4 or other format) showing the board and oscilloscope behaving as expected across the full range of the potentiometer (0v to 5 and back)

Figure A.28: Lab 3 Slide 8

### A.3.2 Given Code

Listing A.5 Lab 3 given code

```
/*
CS431 Spring 2021
Lab 3 Analog I/O
This is one solution, your code may be different
UIUC rfauza2@illinois.edu
*/

// !!!UPDATE!!!!
//in order to use PWM, we need to change the wiring on the board,
//just swap pins 6 and 7 so that pin 6 goes to the LED and pin 7 goes to d7 on the LCD

// include the library code:
#include <LiquidCrystal.h>

// initialize the library by associating any needed LCD interface pin
// with the Arduino pin number it is connected to
const int rs = 12, en = 11, d4 = 3, d5 = 4, d6 = 5, d7 = 7;
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);
const int LED1 = 6;
const int pot = A2;
const int lcdPixels = 16;
//To do:
//Your variables here
// the setup function runs once when you press reset or power the board
void setup() {
//To do:
// initialize LED as an output
// set up the LCD's number of columns and rows
}
// the loop function runs over and over again forever
loop() {
```

### Listing A.5 (cont.)

```
//To do:  
//Your code here (you may want to keep the delay at the end  
of the loop)  
delay(20);  
}
```

### A.3.3 Solution Code

### Listing A.6 Lab 3 solution code

```
/*  
CS431 Spring 2021  
Lab 3 Analog I/O  
This is one solution, your code may be different  
UIUC rfauza2@illinois.edu  
*/  
  
// include the library code:  
include <LiquidCrystal.h>  
  
// initialize the library by associating any needed LCD interface pin  
// with the Arduino pin number it is connected to  
const int rs = 12, en = 11, d4 = 3, d5 = 4, d6 = 5, d7 = 7;  
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);  
const int LED1 = 6;  
const int pot = A2;  
const int lcdPixels = 16;  
unsigned int potValue = 0, oldPotValue = 0;  
float voltage = 0.0;  
int dotPosition = 0;  
// the setup function runs once when you press reset or power the board  
void setup() {  
// initialize digital pin LED_BUILTIN as an output.
```

Listing A.6 (cont.)

```
pinMode(LED1, OUTPUT);
// set up the LCD's number of columns and rows:
lcd.begin(16, 2);
// Print a message to the LCD.
}

// the loop function runs over and over again forever
void loop() {
  potValue = analogRead(pot);
  if(potValue != oldPotValue){
    lcd.clear();
    // display voltages on LCD
    oldPotValue = potValue;
    voltage = 5.0 * potValue / 1024;
    lcd.setCursor(0, 0);
    lcd.print("D:");
    if(potValue > 999){
      lcd.print(potValue);
    }else{
      lcd.print(0);
      lcd.print(potValue);
    }
    lcd.setCursor(8, 0);
    lcd.print("A:");
    lcd.print(voltage, 2);

    // display '.' on LCD
    lcd.setCursor(0, 1);
    dotPosition = (int)(16*voltage/5);
    for(int i=0; i<dotPosition; i++){
      lcd.print(' ');
    }

    lcd.print('');
```

Listing A.6 (cont.)

```
lcd.setCursor(0, 1);  
  // illuminate LED  
analogWrite(LED1, potValue/4);  
}  
delay(20);  
}
```

## A.4 ASSIGNMENT 4 JOYSTICK AND SERVO MOTOR

### A.4.1 Instruction Slides

# Lab 4

## Joystick and Servo Motor

Figure A.29: Lab 4 Slide 1

### Joystick

- The joystick in the Arduino kit consists of two voltage dividers, each of which determines the position of each axis

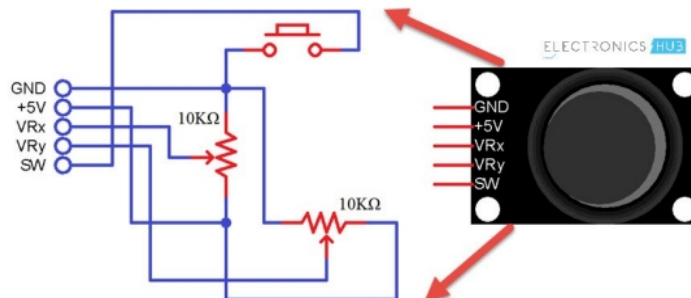


Figure A.30: Lab 4 Slide 2

## DC Motor

- Your kit should contain a DC motor. These motors will turn in the direction of the current applied across its leads, reverse the voltage and the motion direction will reverse
- The Voltage applied across the motor determines the torque (maximum strength it has to push against opposition).
- The Current flowing across the motor determines the motor rotation speed.
- These motors cannot tell the position of their rotating parts

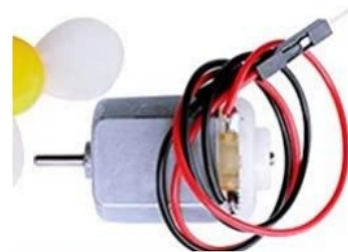


Figure A.31: Lab 4 Slide 3

## Servo Motors

- As opposed to DC motors, Servo motors do not spin continually and their position is digitally controlled by a PWM signal
- The Duty Cycle determines the position of the rotor
- Your board should already be properly connected from the previous labs (brown = ground, red = 5v, yellow = PWM signal)
- See example code at:  
<https://www.arduino.cc/en/Tutorial/LibraryExamples/Sweep>  
p ( see write() function for the values used to control the motor i.e. you will need to convert in your code)



Figure A.32: Lab 4 Slide 4

## Lab 4 Requirements

- Perform joystick calibration: LCD should prompt "Calibrate X min", button 1 press should store that value. Do this for X and Y, for max and min. Calibration and its display should be in a separate function called from setup().
- Make sure holding down the button doesn't trigger the next calibration. Your code should wait for button press and release, and presses should be debounced. Use a different function for each of these (see starter code).
- Display the labeled calibration values on the LCD ("x\_min:33 "x\_max:555, etc"), x on first line and y on second line for 3 seconds (this should all be done in the calibrate() function)
- Clear the LCD and display real time joystick X and Y position, max and min values seen (not calibration). Show X on the first line and Y on the second, formatted: "X: pos,min,max". Displayed range should be between -50 and 50. (have a delay(100) at the end of your loop for better LCD and motor performance)
- Make joystick X (horizontal) position control the Servo motor
- Button 2 press locks motor position and display (use interrupts, see lab 2). Pressing again unlocks. Display locked state with a "U" or "L" at the end of the second line
- Submit to Compass: your code (lab4-netid.ino) and a quick video (lab4-netid.mp4 or other format) showing the board behaving as expected

Figure A.33: Lab 4 Slide 5

### A.4.2 Given Code

Listing A.7 Lab 4 given code

```
/*
CS431 Spring 2021
Lab 4 - Servo and Joystick
This is one solution, your code may be different
UIUC rfauza2@illinois.edu
*/

// include the library code:
#include <LiquidCrystal.h>
#include <Servo.h>

// initialize the library by associating any needed LCD interface pin
// with the Arduino pin number it is connected to
const int rs = 12, en = 11, d4 = 3, d5 = 4, d6 = 5, d7 = 7;
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);
```

### Listing A.7 (cont.)

```

unsigned long debounceDelay = 200;
const int button1 = 10, button2 = 2, LED1 = 13, LED2 = 6, LED3 = 8;
const int x_in = A0;
const int y_in = A1;
int _xmin, _ymin, _xmax, _ymax, xMin, yMin, xMax, yMax, xVal, yVal, xIn, yIn;
int button1State = 0;
int prevButtonStates[2] = 0,0;
unsigned long lastDebounce[2] = 0, 0;
Servo servo;
int servoPos = 0;
int lockedState=0;
unsigned long lastLockDebounce=0;

// the setup function runs once when you press reset or power the board
void setup() {
  Serial.begin(9600);
  servo.attach(9);
  pinMode(button1, INPUT);
  pinMode(button2, INPUT);
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED1, OUTPUT);
  // set up the LCD's number of columns and rows: lcd.begin(16, 2);

  // TO DO: //attach button 2 to the lock function in an interrupt (see lab2 and Arduino
  docs) //call your calibrate function
}
  // the loop function runs over and over again forever
void loop() {
  /*
  * TO DO
  * read X and Y position, display current, max and min to the LCD
  * move servo to reflect X position of joystick
  * do the above ONLY when the board is in an unlocked state
  * we can't use a spin lock inside the interrupt handler, or we could spin forever
  */
}

```

Listing A.7 (cont.)

```
* so we keep track of locked state
* print the locked state on the second line
*/
}

void lock(){
unsigned long currentTime = millis();
if ((currentTime - lastLockDebounce) > debounceDelay) {
lockedState = !lockedState;
lastLockDebounce = currentTime;
}
}

void calibrate(){
/* TO DO:
* calibrate your joystick: prompt user for the joystick input (x min, x max, etc)
* and wait for button press, store value, wait for release, do this for each parameter.
* store the results in variables that start with "_" (see declarations on line 20).
* Variables without the "_" should be used in the main loop to store current readings
* print all the values to LCD and wait 3 seconds
* clear LCD and return to main loop
*/

    int debounce(int pin){
//TO DO
// your debounce code here
return state;
}

int waitForPress(int pin){
//TO DO
//wait until you sense the button press. When you do, debounce it
//it's ok to spin lock here
return;
}
```

#### Listing A.7 (cont.)

```
int waitForRelease(int pin){  
  //TO DO  
  //wait until you sense the button release. When you do, debounce it  
  //it's ok to spin lock here  
  return;  
}
```

#### A.4.3 Solution Code

#### Listing A.8 Lab 4 solution code

```
/*  
CS431 Spring 2021  
Lab 4 - Servo and Joystick  
This is one solution, your code may be different  
UIUC rfauza2@illinois.edu  
*/  
  
// include the library code:  
include <LiquidCrystal.h>  
include <Servo.h>  
// initialize the library by associating any needed LCD interface pin  
// with the Arduino pin number it is connected to  
const int rs = 12, en = 11, d4 = 3, d5 = 4, d6 = 5, d7 = 7;  
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);  
unsigned long debounceDelay = 200;  
const int button1 = 10, button2 = 2, LED1 = 13, LED2 = 6, LED3 = 8;  
const int x_in = A0;  
const int y_in = A1;  
int _xmin, _ymin, _xmax, _ymax, xMin, yMin, xMax, yMax, xVal, yVal, xIn, yIn;  
int button1State = 0;  
int prevButtonStates[2] = 0,0;  
unsigned long lastDebounce[2] = 0, 0;
```

Listing A.8 (cont.)

```
Servo servo;
int servoPos = 0;
int lockedState=0;
unsigned long lastLockDebounce=0;

// the setup function runs once when you press reset or power the board
void setup() {
  Serial.begin(9600);
  servo.attach(9);
  pinMode(button1, INPUT);
  pinMode(button2, INPUT);
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED1, OUTPUT);
  // set up the LCD's number of columns and rows: lcd.begin(16, 2);
  // Print a message to the LCD.
  attachInterrupt(digitalPinToInterrupt(button2), lock, RISING);
  // calibrate();
}

// the loop function runs over and over again forever
void loop() {
  if(!lockedState){
    lcd.clear();
    xIn = analogRead(x_in);
    yIn = analogRead(y_in);
    //convert position value
    xVal = (int)((((float)xIn)/1024)*100)-49;
    yVal = (int)((((float)yIn)/1024)*100)-49;
    lcd.setCursor(0, 0);
    lcd.print("X:");
    lcd.print(xVal);
    lcd.print(",");
    if(xVal < xMin)
      xMin = xVal;
```

Listing A.8 (cont.)

```

}
    lcd.print(xMin);
lcd.print(", ");
    if(xVal > xMax){
xMax = xVal;
}
    lcd.print(xMax);
lcd.setCursor(0, 1);
lcd.print("Y:");
lcd.print(yVal);
lcd.print(", ");
if(yVal < yMin){
yMin = yVal;
}
lcd.print(yMin);
lcd.print(", ");
if(yVal > yMax){
yMax = yVal;
}
lcd.print(yMax);

//control servo
servoPos = (int)((((float)xIn)/1024)*180);
servo.write(servoPos);
}
lcd.setCursor(15, 1);
if(lockedState){
lcd.print("L:");
}else{
lcd.print("U");
}
    delay(100);
}

```

Listing A.8 (cont.)

```
void lock(){
    unsigned long currentTime = millis();
    if ((currentTime - lastLockDebounce) > debounceDelay) {
        lockedState = !lockedState;
        lastLockDebounce = currentTime;
    }
}
```

```
void calibrate(){
    lcd.setCursor(0, 0);
    lcd.print("Calibrate X min");
    waitForPress(button1);
    _xmin = analogRead(x_in);
    waitForRelease(button1);
    lcd.setCursor(0, 0);
    lcd.print("Calibrate X max");
    waitForPress(button1);
    _xmax = analogRead(x_in);
    waitForRelease(button1);
    lcd.setCursor(0, 0);
    lcd.print("Calibrate Y min");
    waitForPress(button1);
    _ymin = analogRead(y_in);
    waitForRelease(button1);
    lcd.setCursor(0, 0);
    lcd.print("Calibrate Y max");
    waitForPress(button1);
    _ymax = analogRead(y_in);
    waitForRelease(button1);
    lcd.clear();
    lcd.setCursor(0, 0);
    lcd.print("x:");
    lcd.print(_xmin);
    lcd.print(", ");
```

Listing A.8 (cont.)

```
lcd.print(_xmax);
lcd.setCursor(0, 1);
lcd.print("y:");
    lcd.print(_ymin);
lcd.print(", ");
    lcd.print(_ymax);
delay(3000);
lcd.clear();

int debounce(int pin){
int state = 0;
if(pin != 10  pin != 2){
return 0;
}
int button=0;
if(pin==2){
button = 1;
}
    Serial.println("DEBOUNCING"+String(button));
unsigned long currentTime = millis();
if ((currentTime - lastDebounce[button]) > debounceDelay) {
state = digitalRead(pin);
lastDebounce[button] = currentTime;
Serial.println("DEBOUNCING");
}else{
state = prevButtonStates[button];
}
return state;
}

int waitForPress(int pin){
while(!button1State)
button1State = debounce(button1);
}
```

Listing A.8 (cont.)

```
Serial.println("Pressed");  
return;  
}  
int waitForRelease(int pin){  
Serial.println(button1State);  
while(button1State){  
button1State = debounce(button1);  
}  
Serial.println("Released");  
Serial.println(button1State);  
return;  
}
```

## A.5 ASSIGNMENT 5 SERIAL COMMUNICATION

### A.5.1 Instruction Slides

# Lab 5

## Serial Communication

Figure A.34: Lab 5 Slide 1

### Serial Communication - Arduino to Arduino

Since we will only have one board available per student, our labs will be limited to Arduino to PC communications over USB, but Arduinos and other microcontrollers can also send and receive serial communication over some of their pins, which is commonplace in embedded applications. This allows for quite complex systems to be designed (like today's cars, airplanes and spacecraft) and does not require a traditional computer to be present.

Read the following page of the Arduino serial documentation:

<https://www.arduino.cc/reference/en/language/functions/communication/serial/>

Figure A.35: Lab 5 Slide 2

## Python GUI Tkinter (optional)

Tkinter is a Python library that allows for the creation of user interfaces for your application. Most of the Tkinter coding will be done for you in the provided code and understanding it is optional, but it would be useful for you to understand it. Tkinter should already come installed with Python 3. If you get errors when importing tkinter, install the latest version of python 3 (<https://www.python.org/downloads/release/python-391/>).

If you still don't have Tkinter and cannot install it, the lab is still doable using the terminal. Comment out the part of the code labeled "with tkinter" and uncomment the part labeled "without tkinter" below it.

Installing Tkinter: <https://tkdocs.com/tutorial/install.html>

Tkinter reference: <https://docs.python.org/3/library/tkinter.html>

Figure A.36: Lab 5 Slide 3

## Python Serial

You will need to install the Python Serial library using the command:

```
$ python3 -m pip install pyserial
```

(If you get a "no module named serial" error, use pip3 uninstall pyserial and install it again using pip3 install pyserial)

Python Serial documentation: <https://pyserial.readthedocs.io/en/latest/shortintro.html>

For serial communication to work, we need to tell Python where to find our device. In your arduino IDE, go to "Tools>Port:", there should be a list of all the USB devices connected to your system. Copy the directory address of your board (labeled "Uno") and the baud rate into Serial.serial() and store it in the "ser" variable in the python code (picture in next slide).

Figure A.37: Lab 5 Slide 4

## Finding Your Board

Copy the board's system address into the "ser" variable as shown

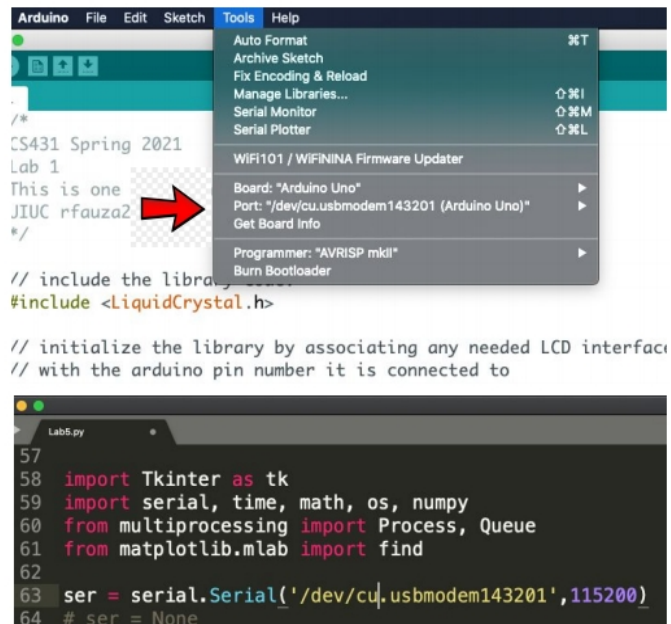


Figure A.38: Lab 5 Slide 5

## Lab 5 Requirements

- Display "Enter your name" on the first line of the Arduino LCD
- Type your name into the Python GUI and send it to the Arduino
- The screen should now display "Lab 5" on the first line, and your name on the second line
- Now have the Arduino continually send readings from the potentiometer (pin A2) to the Python GUI, which should display the value on the screen

Figure A.39: Lab 5 Slide 6

### A.5.2 Given Code

Listing A.9 Lab 5 given code

```
/*
CS431 Spring 2021
Lab 5 - Serial Communication
This is one solution, your code may be different
UIUC rfauza2@illinois.edu
*/
// include the library code:
#include <LiquidCrystal.h>

define MAX_STR 16

// initialize the library by associating any needed LCD interface pin
// with the Arduino pin number it is connected to
const int rs = 12, en = 11, d4 = 3, d5 = 4, d6 = 5, d7 = 7;
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);
const int LED1 = 6;
const int pot = A2;
const int lcdPixels = 16;
const int lcdLines = 2;
unsigned int potValue = 0, oldPotValue = 0;
float voltage = 0.0;
char myName[MAX_STR];
int counter = 0;

// the setup function runs once when you press reset or power the board
void setup() {
  Serial.begin(115200);
  // TO DO:
  // initialize digital pin LED1 as an output.
  // initialize the LCD with the number of columns and rows
  // display message on LCD asking for name
```

### Listing A.9 (cont.)

```
// get name from Python script running on PC by calling readStr() and display it on the
LCD
}
void readStr(char *str)
/* TO DO:
Wait in a while loop that delays 50ms while there is no data available from serial
Once serial is available, break out of the wait loop and read the incoming
string into str[] one character at a time
This should be done in a second while loop that continues while input is available
from Serial, and make sure the string is not larger than the MAX_STR
Include a 2ms delay in the loop to allow byte to arrive in input buffer
Important: make sure to null terminate your string
*/
return;
}
// the loop function runs over and over again forever
void loop()
// TO DO:
// get reading from pot, convert it to the range 0-5v and println to Python (via serial)
// you may keep this 20ms delay
delay(20);
}
```

### A.5.3 Solution Code

### Listing A.10 Lab 5 solution code

```
/*
CS431 Spring 2021
Lab 5 - Serial Communication
This is one solution, your code may be different
UIUC rfauza2@illinois.edu
*/
```

Listing A.10 (cont.)

```
// include the library code:
  include <LiquidCrystal.h>
  define MAX_STR 16

// initialize the library by associating any needed LCD interface pin
// with the Arduino pin number it is connected to
const int rs = 12, en = 11, d4 = 3, d5 = 4, d6 = 5, d7 = 7;
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);
const int LED1 = 6;
const int pot = A2;
const int lcdPixels = 16;
const int lcdLines = 2;
unsigned int potValue = 0, oldPotValue = 0;
float voltage = 0.0;
char myName[MAX_STR];
int counter = 0;

// the setup function runs once when you press reset or power the board
void setup() ]
Serial.begin(115200);
// initialize digital pin LED_BUILTIN as an output.
pinMode(LED1, OUTPUT);
// set up the LCD's number of columns and rows:
lcd.begin(lcdPixels, lcdLines);
// Print a message to the LCD.
lcd.print("Enter your name");
readStr(myName);
  // Serial.println(myName);
lcd.clear();
  lcd.setCursor(0, 0);
lcd.print("Lab 5");
lcd.setCursor(0, 1);
lcd.print(myName);
}
```

Listing A.10 (cont.)

```

void readStr(char *str)
char c;
int i=0;
while(!Serial.available()){
delay(50);
// lcd.setCursor(0, 1);
// lcd.print(counter);
// counter++;
}
while(Serial.available() i<MAX_STR){
str[i]=Serial.read();
i++;
delay(2); //delay to allow byte to arrive in input buffer
}
str[-i]='0';
return;
}
// the loop function runs over and over again forever
void loop()
potValue = analogRead(pot);
// val_a = map(potValue, 0, 1024, 0,999)+1000;
// Serial.print(val_a);
// if(potValue != oldPotValue)
oldPotValue = potValue;
voltage = 5.0 * potValue / 1024;
Serial.println(voltage);
}
delay(20);
}

```