Introduction to the ATmega328P

University of California, Riverside — EE/CS120B Laboratory Assignment 1

Introduction

Welcome to the CS/EE120B Laboratory. While the lecture portion of this class introduces a theoretical framework behind embedded systems design, the laboratory portion provides a means for you to apply the theory you've learned. This lab uses the well-known Arduino UNO featuring the ATmega328P microcontroller. We'll be using the Arduino Elegoo kit available on Amazon. Please make sure to order this ahead of time to prevent submitting late lab(s).

Every lab manual contains three *key* sections: Background, Exercises, and Prelab (separate doc).

- The Background section is *very* important and aims to provide the necessary knowledge on parts/ software to be used for that lab. However, there is no guarantee I'll hit everything in these manuals, so it's important that you attend discussion for the most apt detail. Some components are easier to explain graphically, so I'll provide external links in the Objective section instead and review them in discussion. Use the Background section as a dictionary, looking for concepts as they appear.
- The Exercises section will consist of three-to-four exercises for you to complete. It's important to note that some exercises will build upon previous ones; others will be entirely unrelated. Please read specifications carefully. As of writing this, we plan on hosting in-person demos each week. For you to earn credit for the week's lab, you must demo in-person before the submission deadline. *More details are provided on the syllabus.*
- The Prelab, while not a section in this document, will be posted on the discussion page along with each week's manual. These prelabs are designed to acquaint you with parsing and extracting information from datasheets. You will be tasked with configuring functionality for the following week's lab by learning which register(s) to use, what each bit in said register(s) controls, etc.

By the end of this class, you should have an industry-level exposure to embedded systems. In this effort, it's important to understand that much of embedded systems is built around documentation. Chips these days are jam-packed with functionality, much of which is overkill for most use cases, and this trend is only going to continue as technology improves (as it shrinks). Most embedded platforms provide a method to programatically select functionality that we want to use, and disable functionality that we don't. This process of configuration can become involved and is heavily detailed in documentation. Understanding how to use such documentation is a vital skill which will further your efficiency and independence as an engineer.

In the end, you will hopefully come out with an appreciation for, or, at least, a respect for what embedded systems engineers can do. Be patient, be proactive, and *read* specifications carefully.

Objective

The objective of this lab is to introduce you to the ATmega328P's datasheet, along with the following software concepts & AVR tools:
- Digital Ports & Pins (`PORT[B-D]` & `PIN[B-D]`)
- Data Direction Registers (`DDR[B-D]`)
- Bit-masking & bit-shifting

And the following hardware components:
- Arduino UNO + ATmega328P Microcontroller
- Breadboard
- Light Emitting Diodes (LEDs) & Current Limiting Resistors
- 7-Segment Display

Background

The Arduino UNO + ATmega328P Microcontroller

The ATmega328P was released by Microchip Technologies as an automotive chip (now outdated). It is featured on the Arduino UNO and is the microcontroller this lab will focus on. As implied by the Introduction, there are a variety of features this microcontroller provides. Throughout the course of this lab, you will learn how and when to use these features.

It's important to note, that we **will not be using the Arduino API** as it abstracts away far too much of the underlying technology. The Arduino API is built upon the low-level AVR API that we'll use instead.

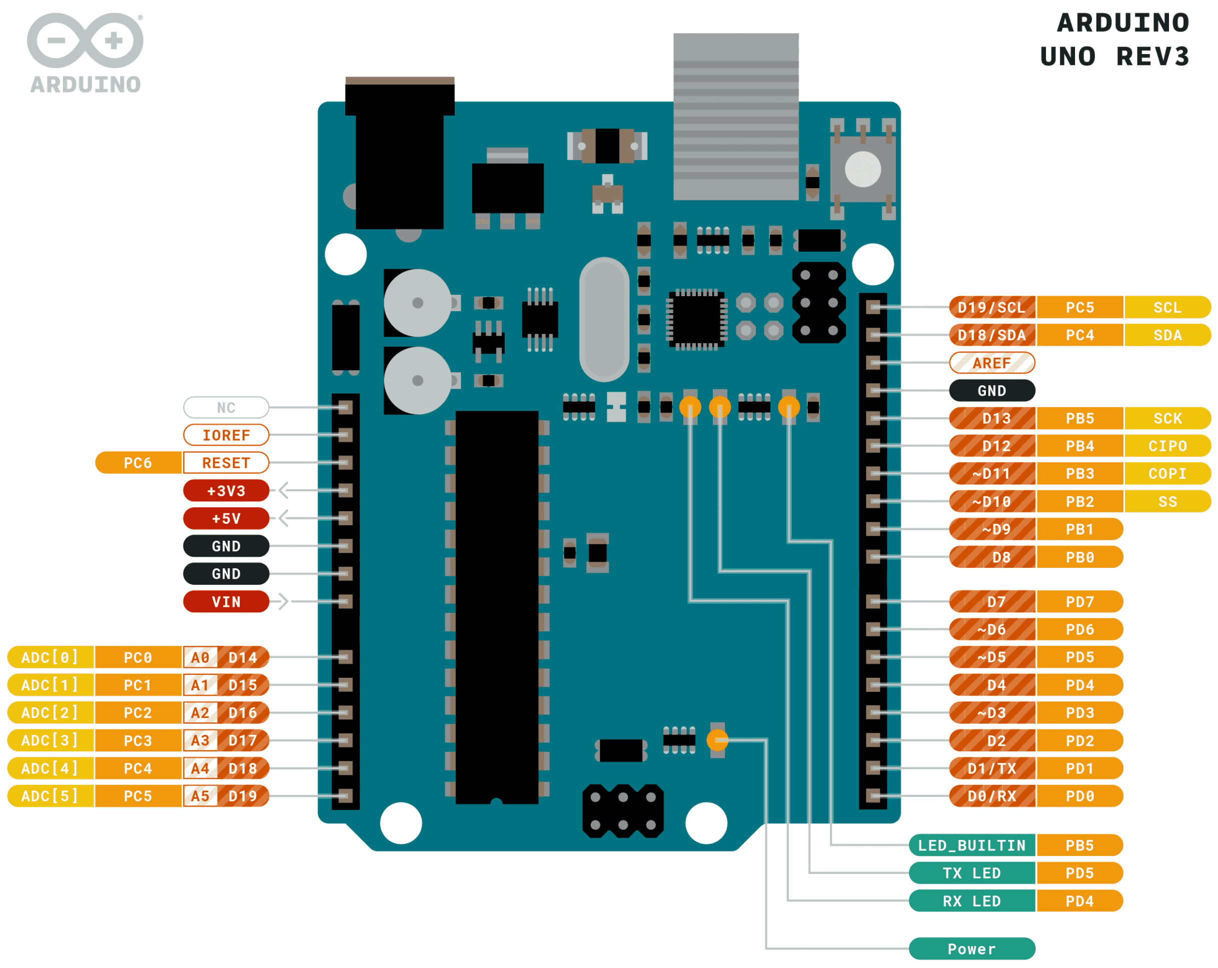

Figure 1: The *pinout* diagram of the Arduino UNO.

Figure 1 is a *pinout* diagram which lists the various functionalities and “names” of each of the pins. Notice some pins have multiple labels while others have only one. I’d like to bring your attention to two types of labels. One of these types are of the form: `D#`, where `#` is a number. If you take a look at the your physical Arduino, you’ll see these are the labels written next to each of its pins. These are the Pin ID’s, and is how pins are addressed in Arduino. We, however, are using the AVR API, not Arduino, and this API uses labels of the form: `PB#`, `PC#`, or `PD#`. Controlling these pins is a little involved, and we describe this in further detail in the next section. Just know that Figure 1 will be useful for translating the Pin IDs used in your physical circuit (Arduino) to the Pin IDs used in your AVR code.

Registers: Ports, and Pins

Recall from CS061, *registers* are storage elements for binary data. Most registers in the ATMEGA serve a similar function—typically storing 1 to 2 bytes. However, a subset of these registers are special, providing additional functionality. Such registers are given unique names and can be accessed directly in code. Of these special registers, some of them are directly connected to the physical pins of the ATMEGA. This means we can not only use these registers to store the results from computation but also communicate that result with the outside world.

The AVR tool chain calls these registers *Ports* and there are three of them: `PORTB`, `PORTC`, and `PORTD`. In your AVR C code, you can write to them as though they’re regular 8-bit variables:

```
PORTD = 214;
```

If you look at the pinout diagram (Figure 1), you’ll see there are pins labeled PD0, PD1, ..., PD7. Each of these these pins corresponds to one of the bits in the `PORTD` register, with PD0 corresponding to the LSb (Least Significant bit) (or bit 0) of `PORTD`, PD1 corresponding to bit 1 of `PORTD`…

So if we assign `PORTD` to the value 214 (which in binary is `11010110`). Pin PD0 is “pulled” `LOW` (displays logic `0`). Pin PD1 is pulled `HIGH`, PD2 is pulled `HIGH`… so on and so forth. Here, you try… should PD5 be pulled `HIGH` or `LOW`? *Hint: Look at the fifth bit from the left.*

So using the `PORTD` register, we can *output* information. What about *reading* information? How can we read data from Port D? To read data, we use the `PIND` register instead. The syntax isn’t much different from writing to `PORTD`:

```
unsigned char var = PIND;
```

Note: The type `unsigned char` specifies two things about the variable `var`. First we know that this variable is 8-bits long (`char`) and that it only stores positive integers (`unsigned`).

You might be wondering why can’t we just use `PORTD` in the same way? What happens if we write `var = PORTD`? Well, you may not face any issues. However, the `PORT` and `PIN` registers are wired differently to the pins. There are subtle implications as to what you’re actually reading. I won’t go into the details here, but they’re explained in the ATMEGA328P’s datasheet.

In general, the rule of thumb when using `PORT` and `PIN` is to use `PINx` to read from the pins of port `x`. And use `PORTx` to write to the pins of port `x`. The terms ‘port’ and ‘pin’ have been slightly abused unfortunately. Please pay attention to the font and capitalization of these terms when reading these manuals. “Port” (without formatting) refers to a **group** of physical pins you can touch on the Arduino; whereas, `PORT` refers to the register, one writes data to in their code. “Pin”, (without formatting) refers to the **indi-**

vidual, physical pins you can touch on the Arduino; whereas `PIN` refers to the register, one reads data from in their code.

Registers: Data Direction Registers (DDRs)

You may have deduced from the previous section, that most pins of the ATmega support bidirectional dataflow. Of course, it's not possible to use pins in both input and output mode. So, instead, we need a way to specify the direction of data flow through a pin. This is the purpose of the Data Direction Registers (DDRs).

There is one DDR for each port: `DDRB` is for controlling the direction of data flow through Port B's pins, `DDRC` for Port C, and `DDRD` for Port D.

Each bit in a `DDRx` register specifies whether its corresponding pin of Port `x` receives data (logic `0` implies input mode) or transmits data (logic `1` implies output mode).

Let's look at an example to better understand this. Say we want to set the lower nibble (bits 3-0) of Port D to input mode, and set the upper nibble (bits 4-7) of Port D to output mode. Then we set the upper nibble of DDRD to `1`′s and the lower nibble of DDRD to `0`′s like so,

```
DDRD = 0b11110000; // the '0b' lets the compiler know the value is in binary
```

Let's look at one more example. Take `DDRD = 0xAA;` What happens here? Think about it. I'll leave the solution in the footer.[1]

Bit-Shifting and Bit-Masking

The previous section should've demonstrated how important bits are in the world of embedded systems. But, if it's not yet clear, then let me say: bits are very important in the world of embedded systems.

Most ISAs[2] don't provide a method of storing & addressing one, singular, bit in memory. The smallest sized type in ANSI C is a `char`, which is 8-bits; and it's also the smallest type we're limited to on the Arduino. But, considering how important individual bits can be in embedded systems, we need a way to address individual bits in our code.

Fortunately, we have several bit-level operators which allow us to manipulate individual bits of a variable. Such operators were likely covered in prerequisite courses like CS061 and CS120A, so I won't go into too much detail—this is, mainly, for review. The first operator to take note of is the bit-wise *and* operator '&'. Keep in mind, this is different from the logical *and* operator '&&'. Bit-wise *and*-ing will *and* each bit of the first operand with its corresponding bit of the second operand. So, for instance, `0b1100 & 0b1010 = 0b1000`.

There is an equivalent operator for bit-wise *or*-ing '`|`'. The previous example, when done with bit-wise *or*, produces: `0b1100 | 0b1010 = 0b1110`.

With these operators, we can strategically craft one of the two operands in such a way so as to extract the value of some bit from the other operand. Let's take an example... Say we have some 8-bit variable `val`. If I want to know what's in the 4th bit of `val`, I can use bit-wise *and* along with a carefully crafted operand

[1]Pins `PD0`, `PD2`, `PD4`, and `PD6` are set to input mode; the rest are set to output mode.

[2]Recall from CS061, ISA is short for Instruction Set Architecture, which describes what operations a given computer architecture can support.

to zero-out every bit except the 4th bit of `val`. It looks like this: `val & 0b1000`. The `0b1000` operand we crafted is called a *bit mask*.

The right and left (bit-)shift operators: << n and >> n are used to shift all bits in a variable to the left or the right by n bits. `0`'s are generally pushed in for the shift operation but not necessarily. Some compilers and ISAs enable you to shift in `1`'s if you right shift a negative value. But, that's not something to worry about for these labs. So, for example, `val << 1`, `val` will be left-shifted by 1. If `val = 0b0010`, then `val << 1` yields `0b0100`.

Shifting has a number of applications. If we wanted to pack multiple, two-bit values into one 8-bit variable. We can use bit-shifting to accomplish this. One can also perform multiplication/division by powers of 2 via shifting. Formally, $x \cdot 2^n$ is equivalent to $x << n$, and $x \div 2^n$ is equivalent to $x >> n$.

There are plenty of other bit-wise operators. We'll cover more of them later. For this lab, however, I suggest taking a look at bit-wise xor in addition to shifting and bit-wise *and*.

Program Structure for Embedded Applications

Your program should always have a `main` function which should *never* return, otherwise, the embedded system is rendered useless. This is the overall structure of a typical application:

```
int main() {

  /* code here executes once */

  while(true) {

    /* code here repeats indefinitely */

  }

  return 0; // return is never reached
}
```

Generally, the space before the while loop is where one configures the microcontroller. The space inside the while loop is what defines the behavior/functionality of your microcontroller. If, for instance, you wanted to blink an LED, then you'd configure the `DDR` and initialize the `PORT` for whichever port you want to use *before* the `while` loop. Inside the `while` loop you'd have update the Port to toggle every so often.

I'd like to turn the page and discuss a notational tool used in writing legible code. Throughout this manual, I've been writing constants in binary. And while it's a nice visualization for small values, for larger values, it becomes illegible. Hexadecimal notation is commonly used amongst embedded systems developers to make code far more compact and legible. Let's revisit the example where we configured the `DDRD` register.

```
DDRD = 0b11110000; // sets the upper nibble to output, and the lower nibble to input
DDRD = 0xF0; // same as 0b11110000, but expressed as hex instead
DDRD = 240;  // same as 0xF0, but in decimal instead.
```

The `0b` prefix tells the compiler that the proceeding value has been expressed in binary. The `0x` prefix tells the compiler, that the proceeding value has been expressed in hex. If no prefix is provided, the compiler assumes the value is expressed in decimal. It's important to note that the machine does **not** literally store

“`F0`” or “240” in memory. What’s stored is `11110000`. That is to say, `0xF0 == 240 == 0b11110000`. Ultimately, they all lead to the same executable once compiled—again, it’s for making code more readable. Each notation has their benefit, it’s important to know when to use which.

Exercises

Required Components (from kit)

- Breardboard + Wires
- Arduino UNO + connector cable
- (x4) LEDs
- (x4) Push buttons
- (x5) 220Ω resistors
- (x4) 1kΩ resistors
- 1-digit, 7-segment display

Schematic

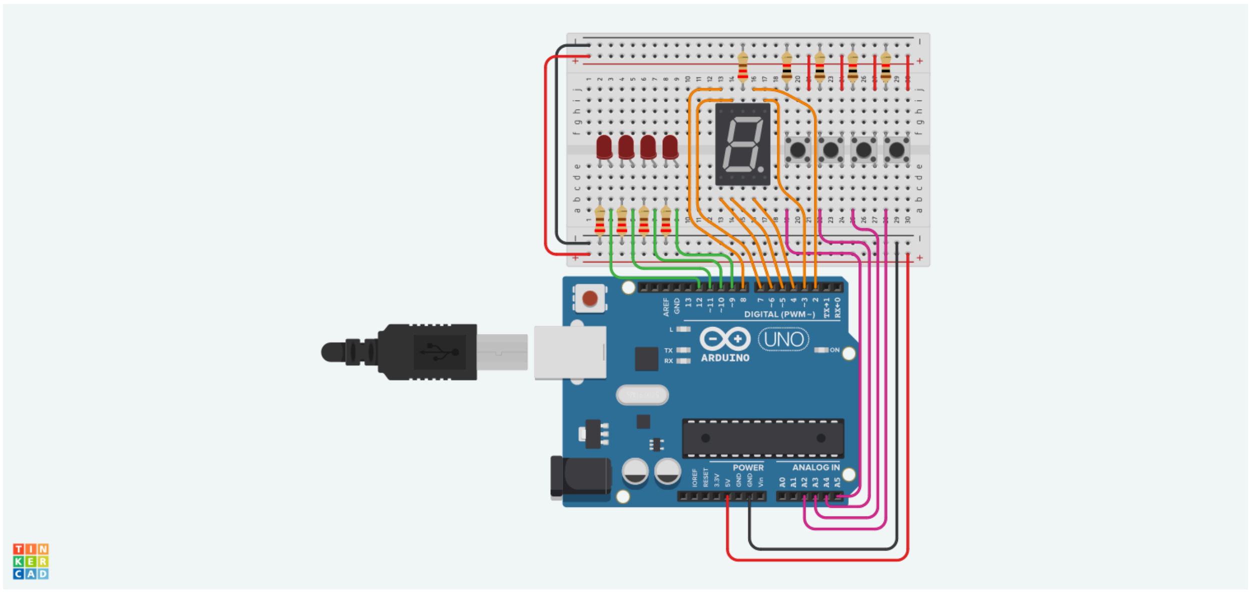

Figure 2: Circuit schematic for Lab 1.

The circuit in Figure 2 supports every exercise. Note that black and red-colored wires convey special meaning: red wires indicate power, and black indicates ground. This convention is used for organization. The other colors were selected to distinguish between groups of components.

Exercise 0

As mentioned before, you'll need to acquire the Arduino Elegoo kit from Amazon. We'll be using this kit for the rest of the quarter. Please be sure to order early so as to avoid submitting late lab(s).

For this class, we won't be using the Arduino IDE, nor are we using the Arduino API; instead, we'll be using VS Code with the Platform IO Extension. VS Code is an IDE, developed by Microsoft, that needs to be installed on your system. Its supported on most Operating Systems including Windows, Mac, and Linux. You can find the download page here. After installing VS Code, you'll need to install a VS Code extension called PlatformIO. This extension comes with all the basic tooling needed to program the Arduino and several other microcontrollers. Instructions for installing PlatformIO can be found here.

After setting up VS Code and PlatformIO, click on the PlatformIO Extension located on the left most column of your VS Code IDE; the icon should resemble an 👽. Click on "Create New Project". The next step is to setup the project. You've got two ways to do this:

1. You can create a new project by following these steps:
   a. Click on “New Project”.
   b. Choose a name for the Project.
   c. Under the “Board” option, fill out `ATmega328P/PA (Microchip)`.
   d. Hit “Finish”, then navigate to `src/main.cpp`.
   e. Replace the `#include<Arduino.h>` header—**remember, you should NOT use the Arduino API**—with `#include<avr/io.h>`.
   f. Replace the remaining functions with the code template provided in the Program Structure for Embedded Applications section. After that, you can proceed to Exercise 1.
2. You can clone this Github repo, then open the `L1` directory through the Platform IO “Open Project” option. After that, you can proceed to Exercise 1. The repo provides some starter code to help you get started. Future lab templates will be posted to that repo.

Exercise 1

Recreate the circuit shown in Figure 2 on your breadboard.

Remember: You should **not** use the Arduino API, use the AVR API described above.

Write an embedded program so that the left-most pushbutton is “connected” to the left-most LED. The next pushbutton is “connected” to the next LED… By “connected”, if the pushbutton is pressed, then its corresponding LED should turn on. If the pushbutton is not pressed, then its corresponding LED should turn off. [`DEMO`]

Exercise 2

Keep the same circuit as the previous exercise, but update the program so that this time, the LED toggles *only* on the button’s down-press (i.e. nothing happens while the button is being held down, lifting up, or idle). [`DEMO`]

Exercise 3

Modify the code to implement a binary counter (from 0 to 15). The four LEDs will represent the value of the counter. The left-most button will increment the count, the next one decrements the count, the next button resets the count (back to 0), and the final (right-most button) divides the value of the count by 2. [`DEMO`]

Note: You will not be asked to demonstrate the behavior of pressing multiple buttons at the same time for these next two exercises.

Exercise 4

For the final exercise, you’ll add another display to the counter: the 7-segment display. Hook up the 7 segment display to pins PD2-7 and PB0 as shown in Figure 2.

Then update your code so that whatever value is displayed in binary (on the 4 LEDs), that same value is displayed in hexadecimal on the 7-segment display. [`DEMO`]

Lab Submission Guidelines

To receive credit for this laboratory assignment, you must:
1. Demo the lab *in-person* to either the TA or Grader during their Office Hours by Monday, August 5th.
2. Submit the source code to Gradescope before 11:59 PM, Monday, August 5th.

Note: Broken parts must be diagnosed by the TA (in-person) ***before*** *Monday, August 5th.*