Introduction to Arduino

Introduction

Just adding a test edit...

With the proliferation of modern microcontrollers, control of electromechanical systems has moved from dedicated circuitry and specialized hardware to more general implementations where a small computer – the microcontroller – is programmed to work with various sensors and actuators using high-level languages (e.g., C or python). One of the primary benefits is that microcontrollers are very flexible, and that flexibility allows the designer to quickly change parameters and try them out without having to significantly redesign the system hardware.

Here, you will be introduced to the Arduino, a low-cost prototyping platform based on the ATmega328P microprocessor from Atmel, fine purveyors of microcontrollers since 1984[1]. The makers of Arduino have taken several Atmel processors and built a platform around them that simplifies programming, communication, power management, and interfacing with peripherals. You will eventually use the Arduino to do many tasks; in this lab and the next, you'll learn about the architecture of the Arduino, write some basic programs, and build a handful of fundamental circuits.

Inventory

All of your materials will be checked out from the Rice 120 Microcontroller Library using the high-tech, bar-code checkout system (props to Danny Voce, TLP third-year, for his work setting this up!). An explanation of the system will be provided during the first lab meeting. Your team will be given a plastic bin to maintain all of your supplies. An official inventory of what you should (and should not!) have in your bin is linked on collab. Keep in mind that this inventory will grow throughout the term as you implement more and more components. Also keep in mind that there are a number of contraband items that you may not keep in your bin – communal items such as DMMs and such. Your team will receive demerits if I find contraband in your bin – accrue enough demerits and it will affect your grade.

The core of your supplies is a SparkFun Inventor's Kit. A complete description of the kit and its components can be found at the SparkFun Inventor's Kit webpage. There is also a reference sheet in the kit, which also has a list of components. There is also a large experimental board with a large breadboard and some switches and knobs. You will also have a red, plastic box to hold minor electronics. You will be given jumper wires and breadboard wires (they have different uses!). Poor wiring techniques will not only make your system more prone to errors, but will lead to sarcasm

[1] and recently bought out by Microchip, fine purveyors of microcontrollers and electronics since 1987.

and derision from your instructor. If your wiring is particularly bad, your instructors will refuse to help you troubleshoot.

The Arduino Uno

Throughout the term, your team will make frequent use of a SparkFun RedBoard, an authorized[2] clone of the Arduino Uno, as well as a wide-variety of electronics: sensors, actuators, and other peripherals. In this class, we will use the terms "Arduino Uno" and "SparkFun RedBoard" interchangeably – the only practical difference is the USB port. In later labs, you will have a chance to "branch out" from the basic Arduino Uno to some more powerful microprocessors (e.g., Atmel's SAMD21 ARM chip).

The Arduino IDE

The Arduino programming language is essentially C++ (a predecessor to java), with a few caveats. First and foremost is that instead of a `main()` as an entry point, all Arduino programs, often called *sketches*, have a `setup()` function and a `loop()` function. When the program starts, `setup()` is run once, and then the program loops through `loop()` indefinitely. That is, you never really "end" a program in Arduino, though you can add an infinite, do-nothing, `while()` loop if you want it to stop doing anything useful.

The Arduino environment provides a number of libraries, some "built-in", some "added-on", to perform a number of common functions. For example, the 328 has several analog-to-digital channels that can be used to read voltages from a sensor (more on this below). Performing the conversion using chip instructions is a bit complicated, so the Arduino environment wraps all the details into a single function, `analogRead()`. The main advantage to this framework is obvious, but the simplicity comes at the cost of efficiency – because they are generic, library functions are generally slower than calling the chip commands themselves. Much of the time, the cost in efficiency is hardly noticed, but occasionally you need to dig deeper into the hardware to extract all the power you can.

Another drawback is that the libraries can be memory hogs. The ATmega328 microprocessor has only a little over 32kB of program memory to work with and only 2kB of RAM(!), and all it takes is two or three poorly written libraries to consume one or the other. On top of that, memory issues can be very hard to diagnose, and the addition of debug statements will only exacerbate the problem. Still, you shouldn't have any problem with the basic projects we'll do in this class. For more advanced systems, we'll use a more powerful processor with more RAM. [3]

The bootloader

Most modern microcontrollers manage programming with something called a *bootloader*, a small program that allows the user to run loaded programs or upload new ones. The bootloader lives

[2] Anyone can clone an Arduino – they're open source – but only those who work with Arduino directly can use the Arduino trademark

[3] Your professor learned how to program on a TRS-80 – look it up – which had a whopping 4kB of memory. Kilobytes. Not Giga. Not Mega. Kilo. Kids have it so easy these days.

on the microprocessor and is automatically run anytime the chip is powered up or reset. The bootloader will then listen to the programming port to see if the user is attempting to load a new program. If so, the bootloader loads the program, stores it, and runs it. If not, it runs whatever program was previously loaded. That is, programs are stored in persistent flash memory, so when you first power up an Arduino, *it runs the most recently uploaded program.* This can be particularly bad if your last sketch did something unwise, such as connect an output pin directly to ground.

Serial Monitor

One drawback of the Arduino IDE is that it has no debugger. Your primary means of debugging, as well as getting information to and from your computer, is through the Serial Monitor. To start the serial communication on the Arduino, you need to add `Serial.begin([speed])` to the `setup()` function, where `[speed]` is the *baud rate* of the communication. Often, a baud rate of 9600 bits/second (bps) is used. If you need to send faster data, you can easily go as high as 115200 bps. To write data to the Serial Monitor, use `Serial.print()` or `Serial.println()`, the latter of which adds a newline character to the end. Note that the Serial class is not very 'smart' – you can't concatenate things easily, so you will often end up putting lots of `Serial.print()` lines in. **If you want your instructor to help you debug something using the Serial Monitor, you must format the output nicely!** This generally means putting related output on a single line, separated by tabs (`'\t'`) or commas. Poorly formatted output will be met with sarcasm and derision.

On the computer side, you can open a Serial Monitor by pressing F12 or clicking on the icon near the upper right of the IDE. Note that doing so will send a reset command to the Arduino, so your program will start over. You'll need to set the communication speed to the same that the Arduino is using. If you get gibberish, you almost certainly have two different baud rates. It's possible to send data from the computer to the Arduino, as well, but we won't use that function in this handout.

With the Arduino Uno, communication to the computer does not happen directly. Your computer is connected to the Uno using a USB connection, but the ATmega328 cannot process USB communication directly. Instead, a second chip – FTDI's FT231XS on the RedBoard – is used to translate from "USB-speak" to a protocol the 328 can understand. This protocol uses something called a `UART`. We'll get into the details later, but the important thing to note is that the `UART` on the 328 is connected to pins 0 and 1 on the Uno. **If you use the Serial Monitor, you will almost certainly not want to use these pins for anything else!**

Tutorials

The following tutorials will cover two standard microcontroller functions: digital output and digital input, as well as getting you started on programming. We'll discuss more about the differences between "digital" and "analog" in class. You may complete these tutorials during the first lab period, time permitting, or on your own time. In either case, **you must complete them as a team.** You should have plenty of opportunities in the first week of classes to meet as a team.

If you'd like more tutorials, the SparkFun website has many to help you get started. You're

welcome to ask your instructor for suggestions. One tutorial that many beginners find useful is how the breadboard is used. You can find that here.

A note on Input/Output

In these tutorials, you will only be using digital I/O. Note, however, that all of the pins on the Arduino can be used for digital I/O, including the "analog" pins A0 - A5. A digital pin can be set as either an `INPUT`, which allows the user to read an external device or as an `OUTPUT`, which allow the user to control an external device. The command `pinMode()` is used to set the mode of the pin to one of three options:[4]

- **OUTPUT**. When configured as an output pin, the pin will either source or sink current *with essentially zero resistance.* It is therefore **very important** to calculate the current that will flow to or from the circuit to which the pin is attached. If the equivalent resistance of the circuit is low, large amounts of current can flow through the pin, damaging the Arduino. *It is extremely important that you never connect an output pin directly to GND or 5V or any other pin that could cause a short circuit.* The command `digitalWrite()` is used to set the pin to a `HIGH` or `LOW` voltage.
- **INPUT**. When set as an `INPUT`, the user can read the pin using `digitalRead()`, which return '0' if the voltage on the pin is low and '1' if the voltage is high. The exact thresholds are left for the student to determine. If nothing is attached to the pin, very small currents, such as the static discharge of your finger, can effectively change the voltage of the pin. When this is the case, the pin is said to "float" – repeatedly reading the pin will produce essentially random HIGHs and LOWs. Another important thing to note is that the impedance (a fancy word for "resistance") of the pin will be extremely high – on the order of $100M\Omega$. This means that essentially no current will flow into or out of the pin when it is declared an input.
- **INPUT_PULLUP**. In this configuration, the pin is set as an input pin (as above), but it is *internally* connected to a $20k\Omega$ pull-up resistor. This allows you to prevent floating inputs without having to wire up additional circuitry. More on that later.

Note that it is possible to switch between modes while a program is running by simply making repeated calls to `pinMode()`. Also, by default, all pins start as inputs, so there is no need to explicitly set a pin to be an input, though it is good practice to do so.

Timers

There are several ways to measure time in Arduino. The ATmega chip has three 8- and 16-bit *hardware timers* that can be used for low-level timing functions, such as pulse width modulation or synchronizing communications. At the lowest level, the *system clock* runs at 16MHz and controls all of the hardware timers, which can be configured for a variety of tasks. Luckily for us, the makers of Arduino have harnessed the power of these hardware timers into easy-to-use software functions. For timing purposes, the most useful functions are,

- `millis()`, which returns an *unsigned*, 32-bit integer that represents the number of milliseconds since the program started. After the value reaches its maximum (what value is that?),

[4]These modes apply to most microcontrollers, not just the ATmega, though the specific methods will vary.

the return value "rolls over" to 0 after the next millisecond. Note that subtraction of two calls to `millis()` works even when the clock has rolled over.

- `micros()` works just like `millis()`, but returns the number of microseconds.
- `delay()` can be used to pause the program for a given number of *milliseconds*. Note that the Arduino won't be able to do anything else while it's in the `delay()` function[5] – it is a *blocking* function – so `delay()` is really only useful for short pauses when nothing critical could happen. In general, `delay()` is frowned upon except for the most basic programs. You will eventually use event-driven programming to time events without blocking execution of the program.
- `delayMicroseconds()` is just like `delay()` but takes an argument in microseconds. Note that it only works in steps of 4 μs.

More information about these functions can be found on the Arduino website. In a future lab, you will use them to write an *event-driven* timer.

Digital output

Procedure

1. Complete Experiment 1: Blinking an LED, the Arduino equivalent of "Hello, World!".

   There are a couple of important things to note:

   - Digital pins (0-13) can be declared as either `INPUT` or `OUTPUT` or `INPUT_PULLUP` (default is `INPUT`), and you can change the pin mode while the program is running. When set to `OUTPUT`, a *digital* pin can be set to either `HIGH` (5 V) or `LOW` (0 V). Note that the pins can only tolerate 40mA – *milli*amps – of current before you risk damaging them, and in most cases you'll want to limit the current to far less. Therefore, *it is essential that you determine how much current is expected to flow through an output pin before you power your Arduino.* **Under no circumstances should you connect an output pin directly to 5V or GND**, as you will almost undoubtedly burn the pin out. It's usually a good idea to include a resistor (at least 220Ω) on output pins where it won't affect the operation of the rest of the circuit, just in case the rest of circuit has a short in it.
   - On that note, what do you think the resistor in the LED circuit is for? Say you measured the voltage drop across the LED and found it to be 1.4V. How would you calculate how much current is running through it?

2. Adjust the delays in the code such that the LED is on for 2 seconds and off for 0.5 seconds and reload the code.
3. Without changing your code, rewire the circuit as shown in Figure 1. **Be sure to remove the GND wire to avoid a short circuit!** How long is the LED on now when you run the program? Why? Is pin 13 sourcing or sinking current?

[5] Interrupts excepted – more on that another day.

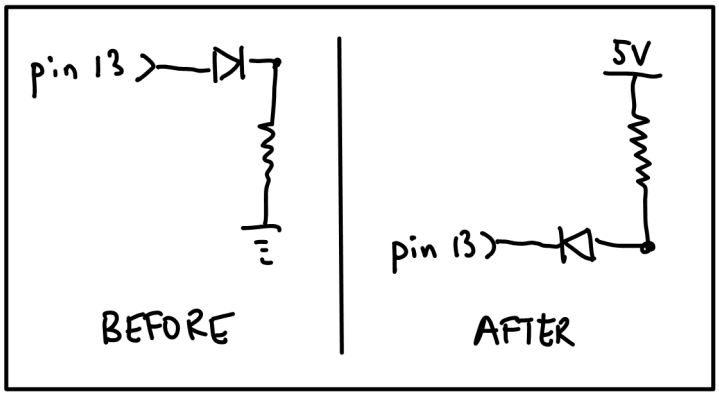

Figure 1: A different LED circuit.

Digital input

Procedure

1. Complete Experiment 5: Push Buttons.

   The most important thing to note is the use of *pull-up resistors*. When a pin is set to input mode, it is connected internally to a very large resistance ($> 100M\Omega$). Because of this, it doesn't take much current to flip the pin from reading `HIGH` to `LOW`; without the pullup resistor, when the button is open (up), noise, stray currents, your finger all would make the state jump back and forth between `HIGH` and `LOW` seemingly randomly. To combat this undesirable behavior, you can employ a pull-up resistor. This loosely connects the input pin to 5V so that when the button is open (up), the input pin will always read `HIGH`. When the button is closed (down), the pin is connected directly to ground, which brings it `LOW` (as an input pin, it is OK to connect the pin directly to GND). We call this an example of *inverted logic*.

2. Note that a small amount of current flows from 5V to ground through the pull-up resistor when the button is pressed. How much?

3. Remove the pull-up resistor. Rub your finger on the breadboard near the button. Does the LED flicker, indicating random changes in the button state? Edit your code and declare the pin mode to be `INPUT_PULLUP`. All should be well again!

4. Finally, edit your code so that pressing the button on pin 2 turns the LED on, and pressing the other button turns it off. For this you will want to use a format like:

   ```
   if button A is pressed => turn the LED on
   if button B is pressed => turn the LED off
   ```

Tutorial Worksheet (5 points)

To be turned in Wednesday, Jan. 24 at the start of class. One per team.

Answers with incomplete or incorrect units will be marked as wrong!

1. Give 10 examples of every day devices that incorporate microprocessors, microcontrollers, or DSPs.

2. What is a *floating input*? How do we combat them?

3. With an Arduino powered at 5V, what is the voltage threshold above which a digital input is *guaranteed* to produce a HIGH reading?

4. What is the largest number that can be returned by `millis()`? **Be exact.** Roughly how long will it take to roll over (from 0 all the way back to 0)?

5. Current through the LED in Experiment 1: _____

6. Current through the $10k\Omega$ pull-up resistor: _____

7. Why do you use a fairly large resistor for a pull-up?

8. Attach your code for turning on and off the LED with the two buttons.