Lab #0: Hello World! and Digital I/O

ECE-121 Introduction to Microcontrollers

University of California Santa Cruz — Fall 2019

COVID-19

As stated in the course documents, we will be making minimal changes to the lab manuals themselves. Please refer to Appendix F for the changes to the lab.

Additional Required Parts

While we have endeavored to make your lab kits complete and sufficient, we did not want to make it expensive by adding in parts you already had from other classes. As such, there are some parts you may need to get from BELS to finish the lab that are not found in the lab kits. For this lab, these are:

COVID Note: The lab kit has been augmented such that the additional parts are either included in the kit or were an optional add on when requesting one. This includes the resistors and capacitors in a very limited selection.

1. Solderless Breadboard and hookup wire
2. Thru hole LEDs (red and green)
3. 7805 Fixed 5V Regulator (possibly a heatsink to go with it)
4. NPN transistor ($\times 2$)
5. Resistors
6. Capacitors

Introduction

Welcome to ECE121 *Introduction to Microcontrollers*! This class will teach you how to program a typical microcontroller at a low-level and on the bare metal. This means there will be no operating system, no HAL (hardware abstraction layer), and no training wheels. We are going to get into the nuts and bolts of the embedded system and learn how to use straight C code to get the micro to do what we want and to interact with the physical world.

This is not a sensors class (ECE167), not a controls class (ECE141), and not a Mechatronics class (ECE118); however we will be utilizing concepts and ideas that are all covered in much more depth in those classes. They are all highly recommended, and in completing them all (including this class) you will be very well positioned to make embedded devices (and will possess highly marketable skills).

We will expect you to know the prerequisite material quite well, as we have too much to cover to spend our precious time going over things that you should already know. Often we will point you to documents from previous classes or other material that we think you should refresh, with the intention that you study these materials if upon quick review you found your retention somewhat lacking.

This lab is a very *quick and dirty* introduction to the tool-chain, working on a simple project, and getting some simple hardware to work with the microcontroller. You will be going over digital inputs and outputs (I/O) and very simple timing based on assembly `nop` commands.[1]

The basic overview of the lab is that you will be driving LEDs, reading ports, and dealing with a simple form of bus contention. You will use this lab to reintroduce yourself to the MPLAB-X IDE (Integrated Development Environment), GIT, the debugger, and some light breadboarding skills.

This lab will divide itself into several parts that will be solved sequentially. To borrow Blue Origin's motto: *Gradatim Ferociter*,[2] you will be developing and testing each required subpart before integrating them into a coherent whole.

In fact, *all* of the labs in this class will be structured in a very similar manner: (i) generate library code, (ii) write a test harness that tests that library, and (iii) write an application using the libraries to do the lab.

> **Info:** The strategy of incremental development is one which we will continually emphasize and reinforce through this and many other classes in your engineering education. Much work has been done analyzing project failures, and the most common problem is that systems integration took too long or did not work because the component parts had not been tested together. While at first glance, incremental development appears to be slow, it is—in fact—the fastest way to accomplish the project. This is especially true as complexity grows and more people are involved. Build (code) a little, test a little, repeat. Break things up into smaller, manageable pieces, and get those working and tested. We will be repeating this *ad nauseum*.

1 GIT and Lab Submission

As in your other classes here at UCSC, you will be using GIT for both source version control (and backups) as well as submitting your code using the GIT commit ID. Git is an industry standard tool for version control, and you should know how to use it as part of your engineering education (for example, the entire Microsoft Windows development is version controlled using Git). You will be using Git for version control on your labs, as well as the method to turn in all of the code in your lab programming assignments.

If you are a bit rusty on GIT, review the *CMPE013_GIT.pdf* document on the class CANVAS website. We expect you to be familiar with GIT, and to be able to do the usual kinds of operations with it. Note that if you are using GIT as you code, every time you commit and push, you have created a permanent copy on the server. This means that you can always return to that exact snapshot of your code, and that you can easily and completely recover it should something happen to your local copy.[3]

[1] The assembly language command, `nop`, stands for "No Operation" meaning that the processor will simply idle for computation cycle; it is called in C with the line: `asm(''nop'')` as an inline assembly command. The advantage of using nops is that the compiler will not remove them from loops even though they do nothing. It is a simple way to ensure timing delays, though it is blocking.

[2] Meaning "ferociously incremental" in Latin

[3] As an aside, a robust commit history is also the single biggest defense against accusations of academic misconduct.

Warning: We track these things in the other programming classes, and we very often see that entire labs are submitted with a single commit and push. This is a very poor way to use GIT. Don't be that student, commit (and push) at every (in)significant increment of coding. It is good practice, and should be a habit. If you are not in the habit, set yourself a timer while working and commit and push every 20 minutes until it becomes a habit. Your future self will thank you for this.

GIT is also integrated into MPLAB-X, such that you can immediately see if files have been committed or not, and do so from the IDE interface. The CANVAS assignment for the lab will contain a textbox to submit your commit ID (which submits your code). There will also be an additional assignment for the PDF write-up of the lab report. This is very similar to how you did it for CMPE13, and is contained within the previously referenced *CMPE013_GIT.pdf* document.

2 Hello World!

Before you can develop any application, you need to be able to debug the system and test each individual module. This is true for this class as it is for *any* microcontroller application. Depending on the environment, which microcontroller you are using, and what toolchain you have at your disposal, there are many options for getting data into and out of your micro. This can range from having a single LED that you can blink, to having some form of terminal I/O you have access to, or to having a full in-circuit debugger (ICD).[4]

In this lab, you will be familiarizing yourself with the toolchain, remembering how to use the tools, and the debugger that you used in CMPE13 (now CSE13E). All programming languages have a "hello world" program.[5] This program is generally as simple as possible and demonstrates that both the code and system runs. While the "Hello World" your did on the UNO32 in CMPE13 prints "Hello World," for this lab we will have a blinking LED to indicate success.

Info: We are using LEDs for the "Hello World" rather than the traditional printout because in the case of an embedded system printing out requires use of the serial port. This gets quite complicated if it is not written for you already, and at this point you are not ready. Note that you will be ready soon, as the serial port communication is central to the next lab.

2.1 Simple Digital I/O

In order to blink LEDs, you will need to be able to control the voltages on a specified pin of the microcontroller. Likewise, in order to read a switch, you must be able to convert a voltage level into a binary value within the microcontroller that you can test with your software.

If you are not familiar with Logic Levels and how they translate into voltages (and why they are voltages) then review Appendix A. This is probably a good idea even if you think you are familiar with them.

[4]Once deployed, the options for verifying that your device is correctly working become much more limited. While some devices will allow for the full debug suite through a specialized connector, it is much more common to have a relatively simple set of "idiot lights" LEDs and a simplified telemetry stream that you can get to either through a wired plug or wireless interface. In this class, we will stick to the simplified telemetry stream (and some indicator LEDs) to give some indication of what is going on in the code internally.

[5]In fact, this is true for all languages because K&R states so at the beginning of chapter 1: "The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages: Print the words *hello, world!* This is a big hurdle; to leap over it you have to be able to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy."

Review your lecture notes and the Family Reference Manual chapter on Digital I/O to understand the structure of a typical I/O pin. Note that they are quite complicated, and that there are some subtleties there (which we will hopefully lead you to discover in this lab).

On the *I/O Shield* are a series of 8 LEDs and 4 button switches. In CMPE13 you wrote code (macros) that could initialize the LEDs, set the LEDs, and read the LEDs. Revisit your old code and rewrite this code. Likewise do the same for reading the onboard buttons. These will become handy in later labs, so make sure to test them thoroughly.

Write C code for your Uno32 that will light the onboard LEDs in response to the buttons. Use your debugger to single step through your code, examine any variables you are using, or anything else it is capable of to help you at this early stage. For instructions and helpful tips on using the debugger, refer to the CMPE13_Lab1.pdf document on the class CANVAS webpage. Again, this should be well known to you, but we allow for being rusty with its use. Get comfortable using the debugger now, as you will be needing to know how to use the debugger well in the remainder of this class.

Congratulations: you have successfully managed to write and compile a (simple) program, load it onto your Uno32, and run it while observing the output. You have just completed a full end-to-end check of your toolchain, hence completing the simple *Hello World*.

2.2 NOPs and Loop Counting

In testing your LED and button code in Sec. 3, you most likely wrote blocking code. Blocking code is a section of code that waits for some condition before moving on. This has the unfortunate side-effect of locking up the processor so that nothing else can happen until this section of code releases; thus the code *blocks* the rest of the code from running.

The way around this is to write *non-blocking* code that does not wait while testing for some condition. Rather the code is atomic and fast, and checks if the condition is achieved and sets an appropriate flag and then exits. Thus the first time that the code executes after the condition is met the response is executed. This makes the code responsive to external inputs and snappy with respect to changes.

Most of the time, making non-blocking code requires the use of interrupts, which we are not going to tackle in this lab. So here we are going to implement pseudo-non-blocking code using `NOPs` to implement delay loops.

NOPs (assembly: no operation) are a simple and easy way to delay the processor for a clock cycle, though obviously nothing else happens on the processor when the NOP instruction is being processed. For our purposes here, you will create a delay loop that uses a for loop and NOPs to have the entire delay be 5ms long (see Listing 1). Use the I/O pins to create a square wave and check on the oscilloscope that you have hit the right delay.

Listing 1: NOP count example

```
    int i;
    for (i=0; i < NOPS_FOR_5MS; i++) {
        asm("nop");
    }
```

You will need to calculate the number of NOPs within the loop to get a 5ms timing. For the more clever among you, you should protest that this is still *blocking* code, as the processor cannot do anything while in the NOP loop. This is correct. Your system will respond, at best, at 5ms intervals. However, relative to human beings, that is quite fast (hence we call it pseudo-non-blocking). Once you have your oscilloscope trace indicating that you have the timing right, include a printout of it (and details about your NOP loop) in the lab report.

Make sure that you encapsulate the NOP delay loop into an appropriately named function (for example, you could name it `NOP_delay_5ms()`), as you will be using this quite often throughout this lab. Having it be called as a function makes it easier to write the rest of the code in the lab.

Info: You might begin to notice that there will be lots of little parts to each lab, and that you will be required to submit code for each of these little parts. This is where having lots of projects becomes quite useful (and you can check them into GIT).

2.3 Improved Hello World!

Given what you did in Sec. 2.1 and Sec. 2.2, you now have the requisite functionality to write a simple application using the building blocks.[6] The application is fairly straightforward: count up in binary on the LEDs, reset when the button is pushed. The pseudo-code algorithm for the improved hello world application is found in Alg. 1.

Algorithm 1: `Improved Hello World Program Flow`

```
initialize all software and hardware modules ;
initialize loop counter and LEDbits variables ;
while 1 do
    write LEDbits to LEDs ;
    increment LEDbits ;
    if button pressed then
        reset LEDbits ;
    end
    if LEDbits > 0xFF then
        reset LEDbits ;
    end
    Delay using NOP loop ;
end
```

As pointed out above, this is still blocking code, but the code releases from the block every 5ms, thus giving the opportunity to both update the LEDs as well as check if the button has been pressed. You can also envisage this as a simple state machine with three or four states. It might be useful to diagram this out and use that as the structure for your code. This would be especially useful if any of the major elements (e.g.: writing to the LEDs) were more complicated and required some larger amount of code and hardware themselves.

Note that when you do this code, with a 5ms delay between loops, the LEDs are counting quite fast. The entire 8 LED count happens in ($255 \times \frac{5}{1000} =$) 1.275s. The lower four LEDs are going to appear to be on continuously even though they are not (you could verify this on an oscilloscope).

[6] In this lab, and in this lab only, we are not enforcing code modularity (that is a .h and .c file with the same name that implements your functionality). While modularity is highly desirable, and on larger projects an absolute necessity, we will get to that later in the labs. If you wish to put all of your code into a module and name things well, go right ahead.

switch		LEDs	
		Red	**Green**
open	0	**10Hz**	5Hz
closed	1	**5Hz**	10Hz

Table 1: LEDs Response to Switch

You should modify your code so that the LEDs increment only once every 250ms (but the button check should occur every 5ms). In your lab report, explain how you modified Alg. 1 to accomplish this. At this rate, the LEDs should take approximately one minute to count all the way up and roll over. Note that at this rate, even the lower LEDs are flickering during the count.

3 Digital Input/Output (I/O)

In the end, the microcontroller is useful only in as far as you can use it to interact with the physical world. The microcontroller can read inputs and drive outputs (very often on the same pin and under software control) in a few ways.

The most common method is digital I/O where the pins read a voltage and interpret it as a logical 1 or 0 depending on the value (see Appendix A). When the pins are set as input, they are "high impedance" and can be considered to not draw any current when reading.

For the output, the pins will be connected to a totem pole circuit that will drive the pin hard to the rail or ground depending on what value is written to the pin. They do have current and voltage limitations (see PIC32MX3XX/4XX Family Data Sheet Section 29, and Appendix A) that must be respected or you will damage the totem pole circuit.

One more form of digital I/O is the *open drain* configuration, where the top leg of the totem pole is disconnected. That is, the pin can pull to ground, but not pull the signal up. The line requires a pull-up resistor external to the micro to be useful. This configuration is quite useful when interfacing to circuits that are not at the same voltage levels as the microcontroller. It can also be used in bus contention (see Sec. 3.2) or for use with other circuits to implement a NOR logic input.

Analog inputs and outputs have their own special configuration and properties; these will be addressed in future labs.

3.1 Driving External LEDs

The LEDs on the I/O Shield have already been set up for you, attached to the correct pins, and have a current limiting resistor on them. Basically, all the work has already been done. In this section of the lab, you are going to be driving an LED off of a 12V supply, but with an output that can only reach $3.3V$. This can also be used for circuits that require higher current than your device can handle.

The schematic for how you will do this is shown in Fig. 1. You will be using an NPN transistor as a switch to drive the LEDs. This is a useful technique when controlling a larger voltage with a smaller one. There are many applications where you need to drive a 12V signal, and in this section of the lab you will learn how to do this. If you are unfamiliar with using a transistor as a switch (in the fully saturated zone) then review the material in Appendix C.

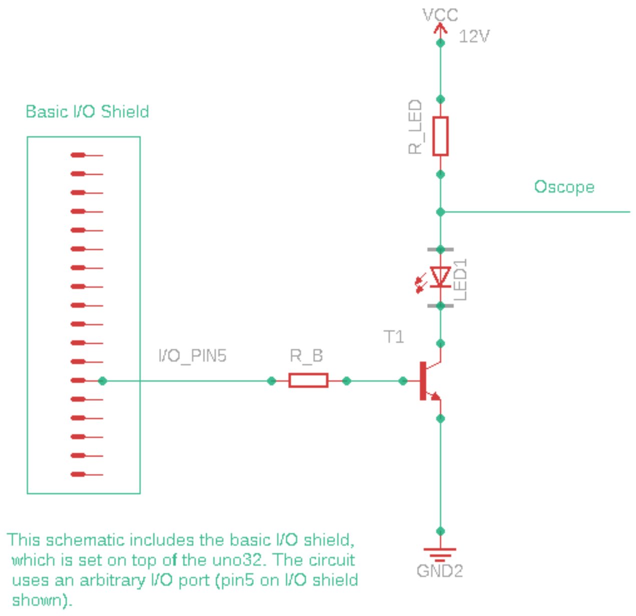

VCC
12V
Basic I/O Shield
R_LED
Oscope
LED1
T1
I/O_PIN5
R_B
GND2
This schematic includes the basic I/O shield, which is set on top of the uno32. The circuit uses an arbitrary I/O port (pin5 on I/O shield shown).

Figure 1: Schematic for 12V driven LED

Size the resistor between the 12V rail and the LED to give you 30mA of current through the LED (this will make it nice and bright). The 5mm through hole LEDs typically have a V_f of approximately $2V$. Choose an LED from your lab kit (or get it from BELS) and measure the V_f and size the resistor so that the *maximum* current is 30mA. Also choose the base resistor, R_b, that you need to drive the 50mA of current through the LED. Use a worst case design philosophy, such that the worst standard 5% resistor will still give you under your current limit.

Include the calculations of your current limiting resistors in your lab report.

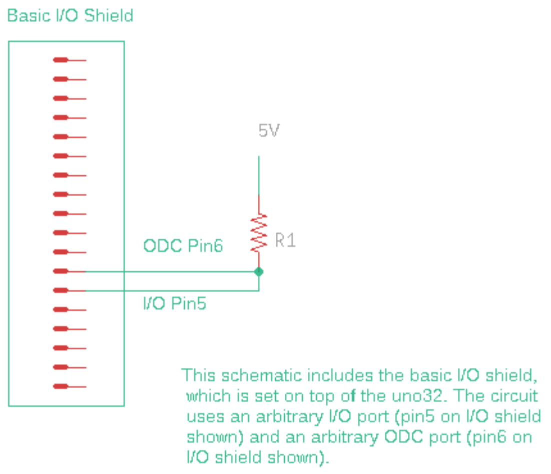
Set up two LEDs of different colors to be driven by two different pins on the microcontroller. You will use your NOP loops to drive one at 10Hz and the other at 5Hz (see Table 1). Hook up the scope to show the 3.3V driving square wave and the 12V driven square wave for both (the 12V should come from a lab power supply). Include the image and your code listing in the final report.[7]

You will also be setting up an external switch into the Uno32 that you will read to alter the behavior of your LEDs. In order to set up the external switch, you will need a lower voltage rail that you can use to power the switch. You are going to make this using a 7805 regulator (note here that we are relying on the 5V tolerant pins) to create a 5V rail for use with the switch. You will also use this regulator in Sec. 3.2. Refer to Appendix B on how to set up and properly filter a regulator.

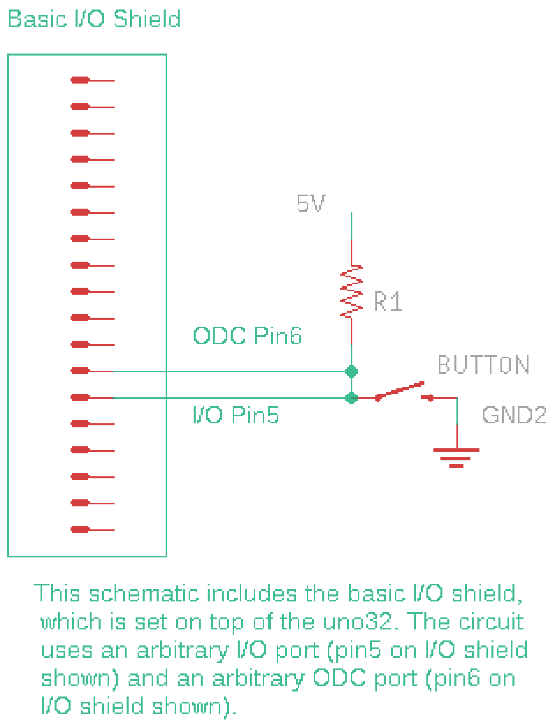
Connect a simple switch to one of your digital I/O pins available on the Uno. See Appendix E for details on how to wire the switch in to the Uno. Make sure that the top of the switch is a regulated $5V$ or $3.3V$. Once you have the switch wired and you can read the switch value (don't worry about debouncing the switch), use the switch to swap the rate of the LEDs. That is, when the switch is up, the green LED is blinking at 10Hz and the red one is blinking at 5Hz (see Table 1). When the switch is down, the green one will blink at 5Hz and the red one at 10Hz.[8]

[7]In order to check this visually, you might need to slow down your flash rate to 2Hz and 1Hz; the human eye can't tell much difference between 5 and 10Hz flashing.

[8]Obviously substitute in whatever colors you decide to use, the red and green are just representative.


Basic I/O Shield
5V
R1
ODC Pin6
I/O Pin5
This schematic includes the basic I/O shield, which is set on top of the uno32. The circuit uses an arbitrary I/O port (pin5 on I/O shield shown) and an arbitrary ODC port (pin6 on I/O shield shown).

(a) No Switch


Basic I/O Shield
5V
R1
ODC Pin6
BUTTON
I/O Pin5
GND2
This schematic includes the basic I/O shield, which is set on top of the uno32. The circuit uses an arbitrary I/O port (pin5 on I/O shield shown) and an arbitrary ODC port (pin6 on I/O shield shown).

(b) With Switch

Figure 2: Bus contention circuit with and without switch

Warning: Be very careful in your electronic hygiene when breadboarding in this class. You are using a 12V supply that will damage the processor if it comes into contact with one of the I/O pins. Double check your connections, and be careful not to have loose wires around. **Never** make changes to your breadboard with the power supply on. Always turn off the power supply before inserting or removing wires.

3.2 Bus Contention

In many communications protocols, there is a process of bus arbitration. That is, multiple devices use the same physical layer to transmit and receive the data, thus all the devices must figure out a way to share the bus without stepping on top of one another.

In every case, this requires the transmitter to read back the bus as they are transmitting to see if, in fact, they (temporarily) own the bus. When a mismatch occurs between what they output to the bus and what they read back, then they know they have lost the bus.[9] This is referred to as *bus contention*.

In order to do this at the physical layer, the transmission pins need to be open collector or open drain (that is that they can only pull the bus to ground, not drive it high). An external pull up resistor will pull the bus high.

[9]Note that there is both destructive and non-destructive arbitration, and if you are interested certainly spend some time understanding how the different protocols work.

Warning: Make sure that you read and re-read the Family Reference Manual section 12 on Digital I/O. Pay special attention to paragraph 12.2.4 on the `ODCx` registers. You will damage your Uno32 unless you have `ODCx` configured correctly.

In this lab, you are going to set this up experimentally and see it in action (and have your software understand and react to it). Fig. 2a shows the circuit you are going to set up. Note here that we are using the fact that the Uno32 has 5V tolerant I/O pins.

Set your output pin to open drain mode (using `ODCx` registers), and drive the signal at a 10Hz square wave while recording the input on the input pin. That is, the output of one pin (in ODC mode) is hooked to the input of another I/O pin. This is commonly called a "loop back." Hook up an oscilloscope and note that the amplitude of the square wave should be 5V. You should also be able to check in software that you are reading in (on the input pin) the same as you are writing out (on the Open Drain output).

Now add in the switch as shown in Fig. 2b. Note that the switch will pull down the bus, while the open drain output will think it is driving to a high state. You are now able to compare your output with the input, and see when you have lost the bus (switch closed). Write an application to create the square wave output and read it back. When the bus is lost, light an LED to signal that the bus is lost. Demonstrate this to the TAs.

4 Final Implementation and Check-Off

For the final checkoff of the lab, you will need to demonstrate the full functionality of all of the sections to the TAs; this can be done all at once, or piecemeal depending on their availability. Note that you will have written several blocks of code to get to this point in your lab. Make sure that your code is well documented and has good comments and style.

As this is the first lab, your blocks of test code should be conditionally compiled using *#define* macros that switch which test is running. This makes it very easy to run the different check-off's.

The final code should initialize all of the hardware and software, and be able to drive the LEDs and respond to the switch. Your breadboard should be **neat and tidy**, and the schematics included in your lab report.

Once you have your code completed, show off the full functionality to your classmates as well as the TA's for check-off. The check-off is the *only* record we have that you completed the lab, make sure you get this done, and it is witnessed by the TAs.

In the lab report, ensure that a student who has already taken the class and is fairly well versed in embedded software would be able to recreate your solution. Include details and algorithms, pseudo-code, flow-charts, and any "bumps and road-hazards" that a future student might watch out for. Please be clear and concise, use complete sentences, and maintain a dispassionate passive voice in your writing.[10]

Congratulations: you have completed your first lab, demonstrated that the toolchain works end-to-end, and you have largely coded it from scratch. Pat yourself on the back, as you have demonstrated that you have the prerequisite knowledge to do well in this class.

[10] English teachers hate the passive voice, but it is used as the primary narration in scientific writing as you want the reader to know what was done, but not be concerned with who did it. In this sense, proper scientific writing is *egoless*. If passive voice is too burdensome for a particular sentence, use the "we performed" or other such construct instead.

Warning: If you took a very long time, or struggled with this lab, it is an indication that you might not be ready for the class. The material will build up fast, and the labs will get much harder. Be mature and realistic about your capabilities; if you are struggling now you will be struggling much more later.

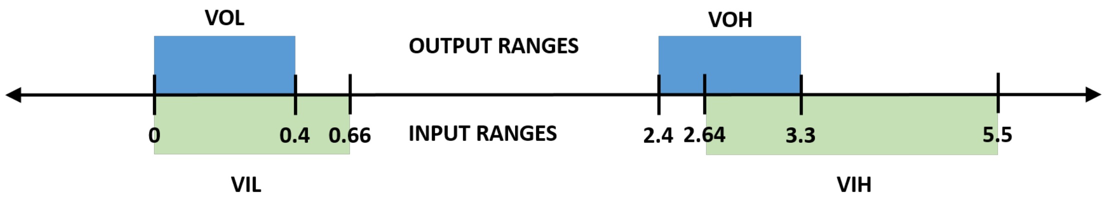


Figure 3: Logic Levels on $3.3V$ Rail

A Logic Levels

When working with computers in general (including microcontrollers), we employ a convenient shorthand to simplify our programming tasks: we refer to everything as binary (values of 0 and 1). Sometimes we refer to these as *HIGH* and *LOW*. This is true even when we are working at the gate level of the computers (as in CMPE12), as well as within the software domain.

However, as with all shorthand, there are details lost in translation that are important. The question to ask is, "What exactly does 0 or *LOW* mean, and how do I know I have it?" Within the computer and on the pins, we do NOT have a simple binary 0 or 1, in reality we have electrical voltages that must be translated to binary equivalents.

The way that this is handled is using *Logic Levels*. Logic levels are agreed upon thresholds such that all compatible devices translate voltage levels consistently. The precise logic levels are dependent on the underlying gate technology (e.g.: TTL, CMOS, HC, etc.). The exact logic levels will depend on the voltage rail of the device used.

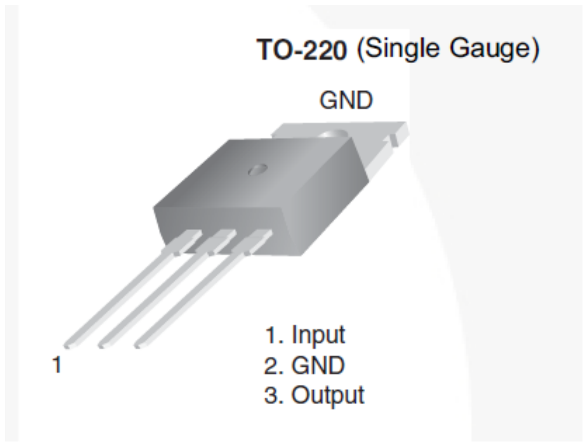
For example, the Uno32 runs on a $3.3V$ rail, and will register a logic level 0 or *LOW* for any voltage between $-0.3V$ and $0.5V$. Note that below $-0.3V$ you will damage the I/O pin. Likewise, a logic level 1 or *HIGH* corresponds to a range of $2.6V$ to $3.6V$. Above $3.6V$ you can damage the I/O pin (although most of the I/O pins on the Uno32 are $5V$-tolerant, meaning that the will not be damaged up to 5.5V and continue to register a logic level 1). This is usually shown in a diagram such as Fig. 3.

There are also guarantees on what outputs the device will produce when writing a 1 or a 0 to an I/O pin. These are listed in the PIC32 Datasheet in Section 29 (Electrical Characteristics).

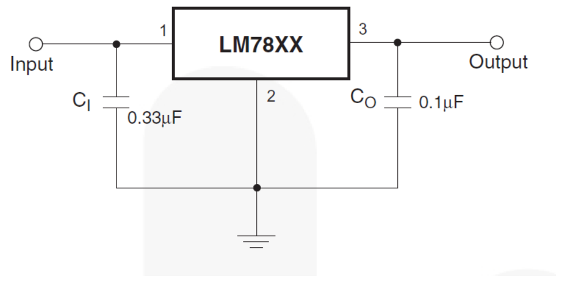
Looking at Fig. 3, the obvious question is "What happens when the input is in between $0.5V$ and $2.6V$?" Does the device read a *HIGH* or *LOW*? The answer is somewhat ambiguous, and not very satisfactory: the device will flip over at some voltage between the ranges between a 0 and a 1, however that exact voltage is unknown, not controlled for in manufacturing, and different for each individual device. Thus every effort is made to keep the voltages out of that middle range and to cross through it as fast as possible.

B Regulators

The basic regulator used in most circuits is a fixed output LDO (low drop-out). This is a three terminal device with an input, ground, and output. The input is the supply voltage, which must be above the output voltage by the dropout voltage. As long as the voltage on the input is above the output + dropout voltage, the output will remain stable at the fixed voltage. Typically a diode is added inline to the input to ensure that if you



(a) TO220 Pin Outs



(b) Filtering Cap Requirements

Figure 4: Physical Device and Schematic for LM7805

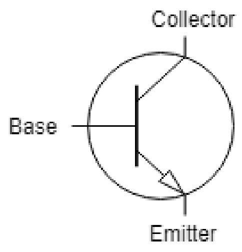


Figure 5: NPN Transistor

reverse polarity on the input and ground, no current will flow (reverse polarity protection). See Fig. 4 for the pinouts for the TO-220 package of the 7805 regulator.

The regulators available from BELS are 7805 (Fixed output at 5V) and can supply up to 1A of current. Note that they have a power rating, and will heat up approximately 70°C for every watt you pull through (they have an over current and thermal shutdown built in). If you are drawing a lot of current through, you will need to add a heatsink.

For stability, the regulator requires both an input filtering capacitor as well as an output filtering capacitor. Typically a 0.22uF cap on the input and a 0.1uF cap on the output will ensure that the regulated voltage stays good even with changes in the load. Adding a regulator and reverse polarity protection to your circuit makes your circuit robust to variable inputs, and errors in hooking up power and ground. Use regulators with reverse polarity protection in every circuit module you make.

C Transistors as Switches

There are many applications where you need to drive a larger voltage load using a small signal, and these are especially prevalent when you are using a microcontroller to interface with the physical world. Given this mismatch between voltage levels, the question becomes how to control the larger voltage source from the smaller one (one can imagine the reverse as well, through in that case a simple voltage divider could also do it).

The answer is the transistor. A transistor (see Fig. 5 is a standard three terminal device, with the terminals designated base, collector, and emitter. In this lab, we will distinguish between BJTs and FETs, and what follows is for BJTs. They come in two flavors, NPN and PNP, which have to do with the semiconductor sandwich that creates the transistor. For more detail on the transistor, including the history and construction, see the Wikipedia article on transistors.

While there are many uses for transistors, in this class we will be using them as switches, and hence we want them fully saturated when on. We will be focusing on the NPN transistor, as this is used for a low-side drive, which is the most common application for microcontrollers. The basic functionality of the transistor is that of a current amplifier. When current flows into the base, a larger current is allowed to flow between the collector and the emitter. Think of it as a current controlled valve, where the current in thin pipe (base) controls the flow of current in the fat pipe (collector-emitter).

It takes both voltage and current to turn a transistor on. The typical base-emitter voltage (V_{BE}) is $0.6V$. This is also the typical forward diode drop, and an easy way to remember this is that the arrow in the BJT symbol looks somewhat like the diode symbol. In the case of driving a large load with a low-side drive, the emitter is usually tied to ground. If the BJT is fully saturated (turned on as most as it can be), then the voltage drop between the collector and emitter (V_{CE}) will be approximately $0.2V$. The last rule of thumb is that the current gain ($\beta = 10$). These relations are summarized in Eqn. 1:

$$V_{BE} \cong 0.6V \tag{1}$$

$$V_{CE} \cong 0.2V \tag{2}$$

$$\beta \cong 10 \tag{3}$$

For example, we are going to drive a 24V load at 200mA using a 5V input to an NPN transistor, determine both current limiting resistors. For the load resistor, we have $R = \frac{V}{I} = (24 - 0.2)/(0.20) = 119\Omega$. Note that there is a standard 5% resistor at 120Ω and at 130Ω; if we use the 130 and it is 5% low, then it would be 123.5Ω and limit the current to 193mA.

For the base resistor, we have an $R = (5 - 0.6)/0.020 = 220\Omega$ using the current gain of 10. Here we wish to take the resistor lower to ensure we have enough drive current, so the next lower standard 5% resistor is 200Ω, which if it were 5% too large would be 210Ω yielding a current into the base of 21mA.

D LEDs and Resistor Sizing

Light Emitting Diodes (LEDs) are one of the most useful devices ever used in embedded systems. They are used as signaling devices that let the user know the status of various things in the system. Some of these are very simple, e.g.: is power on. Other indicators might be much more complex, such as toggling on byte transmission over a communications protocol.

In any case, they are useful and must be driven accordingly. Note that as LEDs have gotten brighter and are now used in both residential and industrial lighting, driving LEDs at very high current has become a necessity.

In order to get the LED to make light, it must have both voltage and current across the terminals. Voltage flows from the anode to the cathode, and the drop across the device (which mostly turns into light) is called the forward drop, or V_f. V_f varies by type and color of LED, with green/red being 1.5-2.5V.

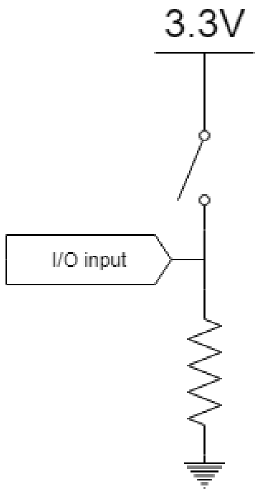


Figure 6: Switch Circuit

The brightness of an LED is directly proportional to the current flowing through the device. Most small LEDs that you will be working with top out around 20mA. Overdriving them is possible, but will result in a much shorter lifespan. Thus LEDs need a current limiting resistor that will prevent you from overdriving and set the nominal brightness of the LED.

The proper current limiting resistor is determined by using Ohm's law ($V = IR$) taking into consideration the voltage drop across the diode. This can get more interesting when LEDs are connected in series.

For example, driving a simple green LED ($V_f = 1.5V$) from a 5V source with a maximum current of 25mA gives an $R = \frac{V}{I} = (5 - 1.5)/0.025 = 140\Omega$. Again, here we want to limit the current to a *maximum* of 25mA, so we choose the next higher 5% standard resistor that ensures that no more than 25mA can possibly flow. A 150Ω resistor give a worst case resistance of 142.5Ω, yielding a maximum current of 24.6mA.

E Switch Wiring

Like the LEDs, the Uno switches are already wired up for you to give a clean signal. There are many times you will need to wire external switches into your microcontroller, and so you need a quick primer on how to connect them.

There are many varieties of switches, but in this lab we will look only at a single pole, single throw (SPST) switch. You can think of this as a normal switch which makes contact when closed, and leaves an open circuit when open.

Fig. 6 shows how the switch should be connected. Note that it is also possible to put the switch on the bottom leg and have the resistor on top. In the diagram shown, the resistor is a *pull down* resistor that drains the pin to ground. When the switch is open, the micro will read a 0, when the switch is closed, it will read a 1.

The resistor should be sized such that you get decent drainage to ground, and not too much current through the switch when closed. A $100K\Omega$ resistor will yield a current of 0.03mA which is good enough to get a clean signal. This is a simple typical value, anywhere from $1K\Omega$ to $100K\Omega$ is fine, use whatever you have handy.

F COVID-19

- For this lab all oscilloscope traces are waived.
- To calculate the 5ms NOP loop you will need to use multiple loops to slow it down to a scale you can see and then divide to get the appropriate number. This will make it slightly less accurate and we will release a nop number after the completion of this lab.
- When driving the two separate LEDs at different rates the speed can be reduced such that it can be more visible.

Acknowledgements

- Images in Fig. 4 courtesy of Fairchild LM7805 Datasheet.