OBJECTIVES

- Understand fundamental input and output (I/O) concepts, in regard to microprocessors and microcontrollers.
- Learn how to manage timing with a software delay as well as with a hardware timer/counter (TC) system.
- Combine I/O and timing concepts to design an LED animation creator program.

INTRODUCTION

It is often desired that a microprocessor be able to interface with an external entity to either receive or transmit data. More generally, all computer systems are designed to utilize some form of **input and output** (**I/O**). Without this, there would simply be no practical purpose for a computer system.[1]

Separately, another important concept in the world of microprocessors is timing. Normally, computer applications require that some thread of execution occur at a specific point in runtime; as a basic example, to blink an LED, a low voltage must be applied to the LED during some intervals(s) of time, and a high voltage must be applied at all other times. In applications such as these, a microprocessor can sometimes manage timing by simply performing some set of meaningless instructions on purpose.[2] In this context, these meaningless instructions constitute what is known as a **software delay**. Oftentimes, a more accurate and advanced approach of utilizing a **hardware timer**, or an independent electronic system that keeps track of time, is chosen. Within the *ATxmega128A1U*, programmable hardware timers exist within the **R**eal-**T**ime **C**lock (**RTC**) and **T**imer/**C**ounter (**TC**) systems.

Overall, understanding how to utilize both I/O and timing is crucial for being able to create any meaningful program for a computer system.

LAB STRUCTURE

In § 1 of this lab, you will explore basic input and output (I/O) concepts, as well as learn to interface your microcontroller with two basic I/O components available on the *OOTB Switch & LED Backpack*: DIP switches and LEDs. Next, in § 2 and § 3 of this lab, you will begin to utilize both of the aforementioned timing mechanisms, that is, both software and hardware timers. Lastly, in § 4, you will design a program that creates LED animations; this will utilize a switch on the *OOTB Memory Base*. This comprehensive application will utilize I/O components as well as timing mechanisms to allow you to create, edit, and display any "8-bit animation" of up to approximately 8000 frames.

REQUIRED MATERIALS

- Atmel XMEGA AU Manual (doc8331)
- *OOTB Switch & LED Backpack Schematic*
- *OOTB Memory Base Schematic*
- *OOTB µPAD v2.0* with USB A/B cable
- *OOTB Switch & LED Backpack*
- *OOTB Memory Base*
- **D**igilent **A**nalog **D**iscovery (**DAD**) with *WaveForms* software
- *Switch Debouncing through Software*
- *The Most Common Use Case for Timer/Counters*

SUPPLEMENTAL MATERIALS

- Adding Additional Program Files To An Atmel Studio Project (GIF)
- Atmel *ATxmega128A1U* Manual (doc8385)
- AVR Instruction Set (doc0856)
- UF *WaveForms* Tutorial
- Digilent *WaveForms* Video Tutorials
- *Using the XMEGA Timer/Counter* (doc8045)
- `lab2 u25 4 skeleton.asm`

---

[1] Relate this idea to the hypothetical thought of human life without sensory communication, i.e., no vision, sound, speech, smell, taste, etc.!

[2] Ideally, the amount of memory used by these meaningless instructions would be small. To do so, some form of looping is generally preferred.

PRE-LAB PROCEDURE

REMINDER OF LAB POLICY

You must re-read the *Lab Rules and Policies* before submitting any pre-lab assignment and before attending any lab.

1. INTRODUCTION TO I/O

Most microcontrollers use *I/O ports* as a construct to share information with other devices. An *I/O port* is simply a collection of pins which can be individually configured for various purposes. Often, an *I/O port* is a conglomerate interface for both an *input port*, a collection of input signals, and an *output port*, a collection of output signals. These pins are referred to as General Purpose Input/Output (GPIO). GPIO ports on the *ATxmega128A1U* are configured and accessed via registers, which are mapped to addresses in data memory space. In most cases, these memory-mapped registers can simply be modified by any instruction which accesses data memory. Sometimes, as is the case in the *ATxmega128A1U*, an individual signal made available by an I/O port has the capability of serving as either an input or an output.

Within the *ATxmega128A1U*, there exist many I/O ports, all of which (collectively, or individually) allow the microcontroller to connect to the outside world. To grant even additional flexibility, each I/O port supports multiple configurations.

NOTES:

- From the perspective of the **c**entral **p**rocessing **u**nit (**CPU**) within the *ATxmega128A1U*, the collection of all I/O ports constitute what is known as a *peripheral system* (a *system peripheral to* the processing unit), and each entity within the system is referred to as a *module*. Throughout this course, a wide variety of other peripheral systems, both inside and outside the microcontroller, will be utilized.
- Most systems within the *ATxmega128A1U*, including the I/O port system, are made to support dynamic configuration. To support dynamic configuration, data storage devices known as *registers* (flip-flop-like components) are utilized. Within the *ATxmega128A1U*, all registers that store configuration information, where these are otherwise known as *configuration registers*, are accessible through the use of dedicated memory locations (within the "I/O memory space") and some appropriate memory-accessing instructions. When configuration memory such as a register, or anything similar, is designed to be accessible through such means, it is said to be *memory-mapped*.
- In the context of the *ATxmega128A1U*, most of the physical pins on the chip package directly connect to I/O port signals. Because of this, most pins on the chip package are named in terms of the I/O port signals! Additionally, instead of having separate dedicated signals for specific peripheral systems within the microcontroller, i.e., those that are not "**g**eneral-**p**urpose **i**nput/**o**utput (**GPIO**)" signals, most signals are multiplexed with those that connect I/O ports. Become familiar with § 33.2 *(Alternate Pin Functions)* within the 8385 manual, which lists all signals that are multiplexed with those that connect to I/O ports (this will become very handy in the majority of your labs, so make sure you are familiar with it).
- If the provided lab kit was assembled correctly, when "backpacks" and "base boards" are plugged into the *OOTB µPAD*, some components, e.g., switches and LEDs, will be directly connected to some I/O port pins on the chip package.

In this part of the lab, you will learn how to interface your microcontroller with the DIP switches and LEDs available on your *OOTB* ***S****witch &* ***L****ED* ***B****ackpack* (***SLB***), where these components represent input and output sources, respectively.

1.1. First, understand how to utilize the memory-mapped I/O configuration registers by reading § 13 (*I/O Ports*) of the 8331 manual

1.2. Study the *OOTB Switch & LED Backpack* and its relevant schematic. Identify any necessary components, as well as where they connect to the *ATxmega128A1U*.

PRE-LAB EXERCISES

i. Which configuration register allows the utilization of a subset of I/O port pins to be configured as inputs (without affecting other pins)? Which configuration register allowed the utilization of a subset of I/O port pins to be configured as an output (without affecting other pins)?

ii. What is the purpose of the SET/CLR/TGL variants of the DIR and OUT registers?

iii. Are the LEDs on the *OOTB Switch & LED Backpack* active-high, or active-low? Draw a schematic diagram for a single LED circuit with the same activation level used on the backpack, as well as one with the opposite activation level. Also, draw a schematic diagram for a **s**ingle-**p**ole, **s**ingle-**t**hrow (SPST) switch circuit, using the same pull-up or pull-down resistor condition utilized on the backpack, as well as another switch circuit using the opposite configuration.

iv. Which I/O ports are utilized for the DIP switches and LEDs on the *OOTB Switch & LED Backpack*?

v. Would it be possible to interface the *OOTB µPAD* with an external input device consisting of 22 inputs? If so, describe how many I/O ports would be necessary. If not, explain why.

To mount the *OOTB Switch and LED Backpack* (as well as any of the other backpacks) onto the *µPAD*, you must match the J1, J2, J3, and J4 headers with the respective ports. The cutout at the top of the backpack (as well as all other backpacks) should allow the reset button on the *µPAD* (labeled S1) to remain visible. See Figure 1 for an image of a correctly mounted *OOTB Switch & LED Backpack*.

1.3. Connect the *OOTB Switch & LED Backpack* to the *µPAD*.

You will now create an assembly program (**`lab2_1.asm`**) to continually, i.e., within an endless loop, output the value of each DIP switch circuit located on the *OOTB Switch & LED Backpack* to a corresponding LED circuit also located on the backpack. More specifically, if an input switch is determined to be closed, the LED located directly below the switch on the backpack must be powered on (i.e., illuminated); conversely, if an input switch is determined to be open, the LED located directly below it must be powered off.

1.4. Write an assembly program (**`lab2_1.asm`**) to allow the *µPAD* to emulate the behavior described above.

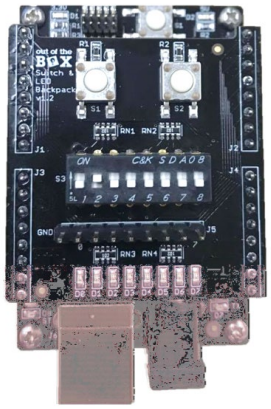

Figure 1: *OOTB Switch & LED* Backpack correctly mounted onto *µPAD*

2. SOFTWARE DELAYS

In this section, you will begin to utilize *software delays*. Software delays are a form of timing delays that are constructed with some (ideally small) set of instructions. Although software delays generally have the potential to be very precise, they prevent a microprocessor from executing other instructions and ultimately cause CPU time to be wasted. Software delays are also extremely non-modular, as they cannot easily be used at other processor clock speeds, and if designed in an assembly language, cannot be directly ported to most other processors.

In general, a hardware timer/counter is preferable to a software delay, however software delays may sometimes be appropriate when a simple delay is needed. You will learn about the hardware timer/counters available within the *ATxmega128A1U* in the following section of this lab document.

To create a software delay, as described above, a (condensed, ideally) series of meaningless instructions should be performed by your microcontroller. The delay length created, in units of time, will be determined by the number and type of instructions executed. (Depending on the series of instructions, the architecture within the microcontroller may add unintended execution time.)

Below, you will create an assembly subroutine to delay 10 ms via a software delay. When determining how many instructions your microcontroller should execute, it would be fair to assume that each one-cycle assembly instruction takes about 0.5 µs, since the *ATxmega128A1U* system clock runs at 2 MHz by default (1/[2 MHz] = 0.5 µs). Remember that frequency (f) is the reciprocal of period (T), i.e., f = 1/T. (Later this semester, you will learn how to change the system clock frequency.)

2.1. Read through the pertinent sections of our course notes, § 3 (*AVR CPU*) within the 8331 manual, and the 0856 (*AVR Instruction Set*) manual to learn about how to write assembly subroutines as well as how to effectively manipulate "stack memory".

2.2. Create an assembly file (**`lab2_2.asm`**). (See the *Supplemental Materials* section of this document for a GIF file that depicts how to add an additional assembly file and set it as the *entry file*, so that you can utilize multiple programs within a single assembly project of *Atmel Studio*.) Within this newly created file, first explicitly configure the "stack pointer" to have an initial value of the highest possible data memory address, as to allow the stack to be of maximal size. (Although the processor automatically performs this procedure, you are expected to perform it explicitly within all programs created throughout this course.) Following this, write a subroutine, `DELAY_10MS`, to simply delay ten milliseconds (i.e., 10 ms), without the use of an additional subroutine, e.g., `DELAY_1MS`. An error allowance of 3% is given. Finally, with the intention to test your delay subroutine, write a main routine (within the same assembly file) to toggle every 10 ms the output of an I/O port pin that is available for probing. To determine which I/O port pins are available for probing, review the appropriate *OOTB* schematics.

NOTE: Within a subroutine, when appropriate, remember to push/pop registers.

To write a subroutine that delays for a specific amount of time, the most efficient technique is to utilize some type of loop structure, calculating the time for each loop iteration.

Delaying for long periods may require nested loops, i.e., loops inside loops (inside loops …). Nesting loops is exactly what is done for counting clock time, i.e., there is a second counter incremented every second; when 60 seconds is reached, a minute counter increments; when 60 minutes is reached, an hour counter increments; etc. Consider a one-hour delay. Start two counters, both at 60, one for minutes and one for seconds. Every second, decrement the second counter; when the second counter gets to zero, restore the 60 seconds in this counter and decrement the minute counter. When both counters are zero, 60 minutes have elapsed.

NOTE: Utilizing many dummy instructions just to waste time is not an efficient delay technique and should be avoided.

2.3. Program your microcontroller with the above assembly file; use your DAD, along with the *Scope* feature of *WaveForms*, to measure the rate at which the specified output pin toggles. (See the *WaveForms* Tutorial, Digilent *WaveForms* Reference Manual PDF, and/or Digilent *WaveForms* Reference Manual Website, if necessary. Video tutorials are also available here.) Make note of the

delay time **initially** measured and include this in the Appendix of your pre-lab report.

2.4. Until the required accuracy is met, fine-tune your delay subroutine. Make notes of any additional delay times measured, including those with error greater than 3%, and include them in the Appendix of your pre-lab report. (Hint: you can try to calculate a precise software delay loop using the clock cycles, but there will most likely be instructions and other delay factors that you did not take into consideration, so you will almost always need to experiment with the number of loops.)

Now, you will create an additional subroutine, `DELAY_X_10MS`, to delay a select multiple of 10 ms specified by the value of a general-purpose register passed into the subroutine. To do so, before the subroutine is called within a program, a specific register must be loaded with a value representing the number of 10 ms delays that should occur. (In essence, this register is storing a parameter, or an argument, of the subroutine.) Ultimately, to perform the necessary delay, the `DELAY_X_10MS` subroutine should call the `DELAY_10MS` subroutine however many times is specified by the register chosen to store the parameter. It is expected that a maximum delay of 2.55 seconds (255×10 ms), as well as a minimum delay of approximately 0 ms, be possible with your delay subroutine.

2.5. Create another subroutine, `DELAY_X_10MS`, as described above.

2.6. Alter your main routine to **toggle** an output pin at a rate of 33.3 Hz (every 0.030 s), i.e., generate a square waveform with a frequency of of 16.7 Hz (period of 0.060 s). An error allowance of 3% is again allowed.

The routine described above should generate a square waveform with a 50% duty cycle, as shown in Figure 2, where 2X in the figure represents the period of the waveform.

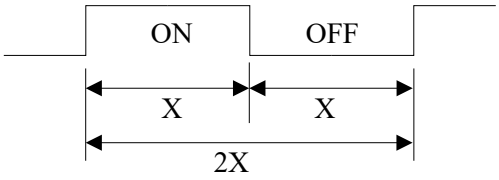

Figure 2: Square waveform with 50% duty cycle, where X is the toggle rate.

2.7. Measure the chosen output pin (again with the *Scope* feature of *WaveForms*) and verify that the **toggle frequency** is 33.3 Hz, i.e., that the overall waveform has a frequency of 16.7 Hz. When the frequency of the waveform is within the given error allowance, take a screenshot, including a frequency measurement of the waveform, and include this image in the Appendix of your pre-lab report.

3. INTRODUCTION TO TIMER/COUNTERS

In this section, you will learn the fundamentals of timer/counter (TC) systems within XMEGA microcontrollers. In general, timer/counters are far more useful than software delays, though normally are slightly more complex to utilize.

3.1. Read § 14 (*TC0/1 – 16-bit Timer/Counter Type 0 and 1*) of the 8331 manual. (The timer/counter systems explained in this section of the manual are those that you are required to use in this lab.) Then, read *The Most Common Use Case for Timer/Counters*. For further supplemental information following these documents, you may also refer to doc8045 (*Using the XMEGA Timer/Counter*).

NOTE: It is possible to split a TC0 system into two 8-bit timer/counters, and it is also possible to concatenate two 16-bit timer/counters into a single 32-bit timer/counter; however, neither of these need to be utilized for this lab.

PRE-LAB EXERCISES

vi. Assuming a system clock frequency of 2 MHz, a prescaler value of 256, and a desired timer/clock period of 30 ms, calculate a theoretically-corresponding timer/counter period value two separate times: once using a form of dimensional analysis, providing explanation(s) when appropriate, and another time using the general formula provided within *The Most Common Use Case for Timer/Counters*

vii. Assuming a system clock frequency of 2 MHz, is a period of two seconds achievable when using a 16-bit timer/counter prescaler value of one? If not, determine if there exists any prescaler value that allows for this period under the assumed circumstances, and if there does, list such a value.

viii. What is the maximum time value (to the nearest millisecond) representable by a timer/counter, if the relevant system clock frequency is 2 MHz? What about for a system clock frequency of 37.444 kHz?

3.2. Referring to *Pre-lab Exercise vi.*, write an assembly program (**`lab2_3.asm`**) to toggle an I/O port pin available for probing every 30 ms, utilizing a timer/counter where appropriate. After writing a complete initial version of this program, use the DAD and the *Scope* feature of the *WaveForms* software to experimentally determine which whole-number digital period value provides a corresponding period with the least amount of error. (Recognize that this whole-number may not be either the truncated or "rounded-up" version of the theoretically-corresponding value.) Provide an appropriate screenshot of the relevant waveform with the minimal amount of error, including its *precise* frequency. (Refer to the *Lab Rules and Policies* for how to appropriately take a screenshot of a waveform as well as for how to appropriately measure a precise frequency.) Additionally, provide within the caption of the relevant screenshot the whole-number value that resulted in a minimal amount of error.

NOTES:

❖ When writing to 16-bit registers (such as PER and CNT), the bytes must be written in order of least to most

significant, otherwise the value will not be updated. This is due to the internal buffering used when writing to registers which represent values greater than 8-bits (§3.11 doc8331).

- In general, the native ATxmega128a1udef.inc file does not provide a direct symbol name for the address of a memory-mapped register that holds a specific byte of some overall value greater than eight bits. For example, although the 16-bit period value within timer/counter module TCC0 is represented with two separate memory-mapped registers, one dedicated to the "low byte" (PERL) and one dedicated to the "high byte" (PERH), there only exists the symbol name "TCC0_PER" which corresponds to the address of PERL. Consulting the offset TC register summary (§14.13 doc8331), PERH is mapped to the address immediately after PERL, and so corresponds to "TCC0_PER+1" (This pattern holds true for most, if not all, other register sets).
- The TC module provides an **overflow flag** (OVFIF in the INTFLAGS register) which is set in hardware whenever the overflow or underflow condition is met (depending on the mode of operation). Use of this flag is essential to the proper operation of the TC system. If the CNT value is compared manually, there is no guarantee that the overflow condition will be caught, as reading and comparing a 16-bit value requires multiple cycles to complete. In this lab the overflow flag will be polled synchronously and must be cleared when an overflow occurs in order to detect the *next* overflow event. (Hint: You should not start the counter until all the relevant registers are configured properly, and you should always make sure the flag is reset after you execute your overflow instructions.)

PRE-LAB EXERCISES

ix. Create an assembly program to perform the same procedure as in § 3.2 but utilize a prescaler value of **4**. Perform everything else described in the section for this new context, i.e., experimentally determine which whole number digital period value provides a corresponding period with the least amount of error, provide an appropriate screenshot of the relevant waveform with the minimal amount of error, including its *precise* frequency, and provide within the caption of the relevant screenshot the whole-number value that resulted in a minimal amount of error. Finally, describe and explain why there may be any differences between the two contexts, i.e., between using a prescaler value of **256**, the value in exercise vi, and a prescaler value of **4**.

x. Create an assembly program to keep track of elapsing minutes with a timer/counter, i.e., design a "watch" that only has a "minute-hand". (Hint: Instead of attempting to configure the period of the timer/counter to directly correspond to sixty seconds, configure the period to correspond to one second, and then keep track of how many times this timer/counter overflows [or underflows, if you wish to configure the timer/counter to count down].)

4. LED ANIMATION CREATOR

In this final part of the lab, you will design a comprehensive application that utilizes I/O components as well as timing mechanisms to create, edit, and display LED animations. (You should have OOTB Memory Base (MB) already connected to the uPAD if you haven't done so already, since this part of the lab will require additional buttons from the MB.) The required program is as follows. (A skeleton file for the program is provided on our course website and is also available through the *Supplemental Materials* section of this document.)

Note: You will utilize all three switches available to you from the uPAD and the Memory Base. Pay attention to all the instructions regarding which switch is used for what purpose in the instructions below.

Upon program start, your application should be in what will be referred to as *EDIT* mode. In this mode, the LEDs located on the *OOTB SLB* should be continually updated with the current logic values of the DIP switches also located on the *SLB*, just as in § 1. However, whenever tactile switch S1 on the ***SLB*** is determined to be depressed (i.e., pressed), the current logic values of the DIP switches should be stored into an 8-bit data memory table only **once**, starting at address 0x2000, where the value stored is to represent a "frame" of the LED animation. At any point during program runtime, the LED animation is defined to consist of only the animation frames currently stored in memory. Data memory addresses up through 0x3FFF should be allocated for the animation.

To guarantee that a value is stored only once per press of the relevant tactile switch, your program should wait for the switch to be released. Separately, it is likely that this tactile switch bounces (upon either a press or release of the switch), as almost all real, non-debounced switches do. To prevent multiple frames from being stored due to bouncing, it is will also be required that your program debounce this tactile switch. For the purposes of this lab, you may either perform this debouncing with a software delay or with timer/counter module. (Within the provided skeleton file, a method of debouncing using a timer/counter module is suggested.)

4.1. Read the pertinent sections of the *Switch Debouncing through Software* document. Note that, although a synchronous method for debouncing a switch with a timer/counter is not described in this document, the techniques described could be easily altered to account for synchronicity.

Your program should remain in *EDIT* mode until tactile switch S2 on the ***SLB*** is pressed. Note that it should not be necessary to debounce or wait for the release of this switch. When tactile switch S2 on the ***SLB*** is determined to have been pressed, what is referred to as *PLAY* mode should commence.

In *PLAY* mode, the entire LED animation currently stored in memory should begin to play sequentially on your LEDs, starting from the initially stored frame. The rate at which each frame of the animation changes, or the "frame rate", should be 5 Hz. (An error allowance of 3% is given.) At rates faster than 10 Hz, you would have a hard time verifying the animation, but could use your DAD's *Logic Analyzer* to verify the animation frames. Observe how changing the frame rate affects the "smoothness" of the animation.) Although software delays could be utilized to accomplish this frame rate, you **must** use a timer/counter. Upon reaching the end of the animation, the LED pattern should restart, and continue to be output until tactile

switch S2 on the ***OOTB MB*** is pressed; after it is determined that this tactile switch has been pressed, the animation should stop, and the program should return to *EDIT* mode. Note that it should not be necessary to debounce or wait for the release of this switch.

Upon returning to *EDIT* mode, the program should function just as it did initially, although the LED animation previously stored should be retained, i.e., if tactile switch S1 on the ***SLB*** is pressed, another frame should simply be stored to the end of the previous LED animation. Thus, the program shall never terminate, and the only way to reset the LED animation should be to either re-program your microcontroller or to use the on-board reset button (labeled S1 on the ***µPAD***).

Since you will need to debounce tactile switch S1 on the ***SLB***, it will first be necessary to measure the amount of time that the switch bounces. To more easily measure the bouncing, you should perform measurements without the backpack connected to the *µPAD.*

4.2. If necessary, remove the *OOTB SLB* from the *µPAD*. Once the backpack is isolated, connect the appropriate voltage supplying pin from your DAD to one of the "3V3" pins on the Switch & LED backpack, and connect the internal ground pin from the DAD to the "GND" pin on the backpack. Following this, provide 3.3 V to the backpack by using the *Supplies* feature within *Waveforms*. (If your DAD/NAD's software is incapable of providing 3.3 V, it is also tolerable to power your backpack with 5 V.) Next, use your DAD and the *Scope* feature within *Waveforms* to record tactile switch bouncing and to determine an appropriate delay time. Refer to the available schematics, if necessary. Submit some screenshot(s) depicting the two relevant bouncing waveforms: one for the press of the switch, and one for the release of the switch.

If you do not see bouncing, try pressing the tactile switch in the following way: slowly, sideways, and fast. If you **STILL** do not see bouncing, then use one of the PORTA switches (like the ones from 3701) to see (and record for your lab report) bouncing when the switch goes from one position to the other and then back to the original position. Use these bounce results to calculate your delay values. Note that if your tactile switch does not bounce today, it may bounce tomorrow or later in the semester.

NOTE: Many configurations within the *Scope* feature should be able to capture bouncing, although it may be helpful to use the *Repeated, Normal* mode, a trigger with a level of around 2 V, an initial timebase of around 25 ms (zooming in when determining if a bounce occurred), and any other appropriate configurations.

4.3. Create the LED animation creation program (**`lab2_4.asm`**) described above. A skeleton file for the program is provided on our course website and is also available through the *Supplemental Materials* section of this document. *Your program should not be susceptible to tactile switch bouncing.*

PRE-LAB EXERCISES

xi. It is stated above that, in the relevant context, it should not be necessary to debounce (nor wait for the release of) either tactile switch S2 on the ***OOTB MB*** or S2 on the ***SLB***. Why is this so?

xii. Provide a scenario in which the above program would experience unintended behavior due to tactile switch bouncing.

PRE-LAB PROCEDURE SUMMARY

1) Answer pre-lab exercises, when appropriate.
2) Familiarize yourself with I/O ports in §1.
3) Learn about software delays in § 2.
4) Become introduced to timer/counter (TC) systems in § 3.
5) Measure tactile switch bouncing and design an LED animation creator in § 4.