Chandler Janzen
Rev 3

CME331 – Lab 2

Microcontrollers
Timers and Mechanical Switches

Safety: In this lab, voltages used are less than 15 volts and this is not normally dangerous to humans. However, you should assemble or modify a circuit when power is disconnected and do not touch a live circuit if you have a cut or break in the skin.
Preparing for emergencies protects our lives and property. A classroom emergency posting is located in each classroom and lab near the main door of the room. Students are advised to review and be familiar with this classroom emergency posting and the College of Engineering Emergency Response Plan (ERP) The building must be evacuated when an alarm sounds for more than 10 seconds.

Objectives: By the end of this lab, you should have learned the following:

1. Become Familiar with a variety of debugging techniques.
2. Know how to use the debugger within CCS.
3. Know how to unlock and use SW2.
4. Know what blocking code is and why it is problematic.
5. Become familiar with techniques to avoid blocking code.
6. Become familiar with software-based timers.
7. Know how to configure a hardware timer from the datasheet.
8. Know how to use a hardware-based timer without interrupts.
9. Know the difference between SysTick and a General Purpose Timer Module.
10. Different configuration options for the hardware based timer.
11. Become familiar with debouncing techniques.

Expectation: Getting the lab done is good. Getting expected results is better. Learning how to work with MCUs properly so that your code and work is modular, expandable, reusable and robust is best. Although these labs are to be done individually, you should envision yourself working on a small portion of a larger project and design team.

Code Submission: Always create your CCS projects with your NSID(s) in the project name (your NSID should be in the 'tree' on the left of CCS). Generic naming of projects (such as 'Lab 1' may result in a grade of 0. Always submit the **entire** CCS project folder (which should already include your NSID in the folder name) with relevant outputs electronically in a zip file (named <NSID>-<Lab#-Part#>) for each part (if applicable) through Canvas/BlackBoard no later than a week after your lab. Enhance the readability of your code by adding explanatory comments wherever appropriate. Always include your name, NSID, and date at the top of each code file. If you are unsure about what is expected, ask your instructor.

Working with a Partner: If you've worked with a partner, please indicate your partner's Name and NSID at the top of your PDF Document, as well as at the top of any source files (as a comment). Only one submission per pair is required.

Late Policy: Each lab is due one week after the scheduled lab time. Every student is granted 2 days late submission without penalty and 10% per day after; however, all submissions must be in prior to the start of the next lab.

1 Introduction

This lab will take a look at different debugging and troubleshooting techniques. It will also take a closer look at using unfiltered mechanical switches as inputs to the MCU, and the problems that exist in them.

2 Background

For your convenience, here is a link to the MCU Datasheet: TM4C123GH6PM Microcontroller Datasheet

2.1 Debugging

Learning how to troubleshoot your code is a skill that is developed, not a skill that is taught. This section will help you explore your options to help you narrow down problems. As you get more experienced with development, you will clue into problems faster each time. Knowing how to use tools and techniques will also significantly improve your debugging time.

2.1.1 Debugging Tools

A common question that I'm asked by junior engineers is 'How do I choose an MCU for my design?' This is a difficult question to answer - I'll add more info later in this course, but criteria should always be: what debugging tools are available for development with this device? Debugging tools serve two purposes: first, it's a means to program the MCU for prototype purposes. Second, it's going to usually be the best/fastest means of troubleshooting your designs. This, in turn, will save you countless hours and companies big dollars.

There are debugging tools that are vendor specific, such as Microchip's ICD 5 . This In-Circuit Debugger (ICD) will work for almost all of microchip's product line (1000's of MCUs). There are also generic (and open-source) debugging tools that tend to have a much wider range of support, such as these JTAG debuggers . There are advantages and disadvantages to both tools, but the trade-off usually involves cost and support/features.

For starters, an in-circuit debugger is a good choice. It allows you to run code directly on an MCU, and get feedback with register information and (typically) gives a full-featured debugger with start/stop/pause functionality. There are also other different types of tools available. There are sometimes simulators built into some of the IDEs. Simulators can be useful to get started, but have difficulties implementing real-time situations (but are cheap - often free!). Another alternative is an In-Circuit Emulator (ICE). These ICE tools are much more advanced versions of ICDs, where the hardware is emulated by a device, and debugging features and speeds are greatly improved (but the price of them also increases).

Similarly, as in the case of these labs, vendors often offer development boards. These boards come with pre-configured hardware that enable you to test an MCU and it's capabilities/peripherals/features in a very quick time-frame. These development boards often include the necessary hardware to program the device, and most of them even offer debugging capabilities without the use of an external tool. Additional hardware allows for quick development of USB/Ethernet interfaces, and other peripherals that would normally require a significant amount of design time to get running. The launchpad that we are using for this course does exactly that - allows us to program the TM4C123, and use a debugger through the CCS interface with easy access to some of the peripherals.

Development boards are a quick way to get started on a prototype, **but are not a good choice to be used in a production-ready final design!** I have to mention this, because I've seen it happen.. too many times. The hardware configuration is undoubtedly not going to be optimized for the design, and production costs will almost always exceed that of your custom design (in most cases, unless production quantities are really low).

2.1.2 Debuggers

Once you have a configuration that supports debugging, you need to learn how to use the debugging interface. You *should have* used a debugger in a previous computer science course to some capacity. Most debuggers have the same basic functionalities, although some have some 'nice' features.

For starters, there is a good tutorial on TI's website: Read this starting at section 7.3: Launching a Debug Session. I've also posted a tutorial in the first lab. **Make sure you go through the tutorial.** In particular, make sure you learn how to:

1. Step through your code line by line (Step in, Step out, Step return).
2. Set a breakpoint and run until you reach that line.
3. View variables when paused.
4. View registers when paused.
5. View expressions when paused.

Note: You cannot properly view pointer variables (as variables or expressions) in the debugger. This includes your symbol definitions for registers. A trick to get around this is to create a variable and assign it to the value at the pointer. You can then view the variable in the debugger. Also, this debugger gives direct access to register information from the register view. There are many more features that you can learn. Mastering the debugger will prove to be the most valuable debugging tool at your disposal. If you have questions, ask!

2.1.3 Debugging Techniques

The first strategy in debugging a problem is to always: **Narrow down the issue to a single line/statement/procedure.** This is why a debugger is unrivalled at troubleshooting problems - it makes it so easy to do that! You can also help yourself by compiling often - the more code you add in-between compiles, the harder it can be to narrow down a problem. The second strategy in debugging is to **eliminate all assumptions made.** Did you assume an initial value? Did you assume the size of a data type? Did you assume what a bit in a configuration register was for? Never make assumptions, and always write your code such that no assumptions are made.

There are situations where the debugger may not help you solve your problem, or perhaps you are working with a device in which there is no debugger (hopefully not..). Here are some extra techniques that you can use to help troubleshoot your issue:

- Get an LED to work. In strategic places in your code, set the LED and visually verify that your code reached the statement. *Don't forget that the MCU runs quickly - toggling an LED can easily be missed if toggled too quickly.*
- For some designs and debugging tools, you can still print to a console. Don't have a console to print to? Maybe you have an LCD or GLCD in your design/development board that can work as a good substitute.
- Don't rule out hardware problems! Pull out that multimeter or oscilloscope/logic analyzer and make sure everything is working properly. A logic analyzer for this development board is a great substitute for an LED. You can even output binary values/information to a port and view it on your logic analyzer.
- Make copies of your code that you *know* are working. You can quickly rule out hardware issues if you run a program that you know works. Always make copies of your code - a good revision control system can aide you greatly.
- Narrow down the problem by reducing your code. Start commenting out blocks of code to narrow the problem. Once you are back up and running, start adding code back in *one block at a time.*
- Use serial communication to output information (UART, SPI, I2C, RS232, etc.). You haven't learnt how to enable these peripherals yet, but once you do you will have a great tool at your disposal.
- Read the Errata. The worst-case scenario: you are working diligently on a design and a problem creeps up, and you can't seem to figure out what's causing it. Don't forget that the design of an MCU is not perfect either. Mistakes make it through into productions. These mistakes are often documented, especially if the MCU is not (relatively) new. Hopefully the Errata suggests a work-around for the problem. This is why it's not always recommended to use the latest and greatest MCU in a new design, and to check the Errata before selecting the MCU - I've seen Erratas that are longer than the datasheet! There are several Erratas that you may have to read:
  - → The Errata for the specific MCU such as this one for the TM4C123x.

→ The Errata for the datasheet such as this one for TM4C123x datasheets.

→ The Errata for the Family of MCUs such as this one for Cortex-M4 MCUs.

→ The Errata for the compiler such as this one for v20.2.1.LTS .

2.2 Locked Registers and Unlocking SW2

Some registers in the MCU are 'locked'. This is to prevent the accidental modification of an important setting. For example, the MCU communicates with your computer through two pins. What would happen if you or your program accidentially reconfigured those two pins such that they can't communicate any more? ***(hint: you brick your launchpad!)*** To assist in the prevention of such potential disasters, the MCU has certain registers or portions locked. While locked, writes to those bits will be ignored. To unlock, you must go through a specific sequence.

Once a register is unlocked, it is imperative that you follow the good coding practices that you learnt in Lab 1:

- Only modify the bits that you are actually using through the bit-masking techniques outlined in Lab 1. (*If you didn't learn when* ***not*** *to use the assignment operator, now is a good time to go back and review.*
- Read the datasheet and understand what the bits do that you are changing.
- Read the schematics to make sure the bits you are changing are not already used.
- Re-lock the MCU immediately after modifying the registers.

Read Page 205 to learn about why registers are locked and how to recover a bricked launchpad.
Read Page 656 to learn about Commit Control.
Read page 684 to learn how to unlock/lock the MCU.
Read Page 685 to learn how to use the commit register to allow a change.
Read Page 1329 to see which pins/bits are locked.

Follow this order when modifying a locked register:

1. Unlock the MCU.
2. Modify the Commit Register for the bits that need to be changed.
3. Modify the Register what was previously locked.
4. Lock the MCU.

2.3 Software Timers

You may have noticed in Lab 1 that the delay function (that counts to some large number) takes a long time to process. While the MCU is processing that delay function (which is a 'blocking' delay), it cannot process any other code, such as checking for a button press. This severely limits the expandability and usability of our design. As an alternative to the blocking code used in Lab 1, we can develop a software based timer that allows for additional actions and processing instead of blocking for an event to occur. Let's review Lab 1's blinking LED:

Example 2.1 (*Blocking Code*)

```
while(1)
{
  toggleLED(); //This is quick and efficient - 'non-blocking' code
  delay(); //This is inefficient and takes too long to process: 'blocking' code
  checkSW1(); //This check only occurs once every ~0.5 seconds.
}
```

To eliminate the delay() blocking function call in Example 2.1, we need to have a means of keeping track of time. There are many ways to do this in an MCU - we could have an external hardware timer, and internal hardware timer, or we can make a software based timer. For now, let's look at making our own software based timer.

Example 2.2 (*Software Timer*)

```
unsigned int myTimer = 0;
while(1)
{
  myTimer++; //increment by one each time we go through the while loop.
}
```

Example 2.2 shows how we can keep track of time. One issue with this method is that it is not accurate. How much (real) time passes with each increment of myTimer? A variable amount - any code addition in the while(1) loop will extend the duration inbetween counts. However, it can still be used as a reference to time; thus, we can blink our LED and not have to worry about missing our switch press as shown in Example 2.3:

Example 2.3 (*Software Timer*)

```
#define DELAY_AMOUNT 100000
unsigned int myTimer = 0;
while(1)
{
  myTimer++; //increment by one each time we go through the while loop.
  if(myTimer > DELAY_AMOUNT) //This is a non-blocking check - it's fast to do!
  {
    toggleLED(); //Toggles every 100000 cycles of the while(1) loop
    myTimer = 0; //reset timer
  }
  checkSW1(); //This now gets checked very often.
}
```

This technique to avoid the blocking code from Lab 1 is a good start, but has the inherant problem of not being expandable or modular. If we want to toggle the LED at 2 Hz, we would need to adjust our DELAY_AMOUNT every time we add more code into the while(1) loop. The resolution and consistency of the timer also decreases with an increase in the complexity of the design.

Tip: You can have as many software timers as you want. You can also use one timer for many different tasks by creating a simple Finite State Machine (FSM). You may find a stopwatch such as this one useful to get your values in the correct range.

2.4 Hardware Timers

Hardware based timers are built into almost all MCU devices; they are likely the single most valuable asset to a design engineer. They greatly improve the capabilities of the MCU and allow for a relatively simple and slow device to accomplish a great number of tasks. You'll find specific information on the general purpose timers available in the TM4C123x MCU on page 704 of the datasheet.

2.4.1 Types of Timers

This MCU has four different timers available to the user. First, there is the system timer (referenced by SysTick in your course notes). The system timer (page 123) is intended to be the heartbeat of an MCU that is running a simple real-time operating system (RTOS). This timer is extremely reliable and simple to use, but has less configurable options available to the user. For the labs in this course, we will not be using an RTOS and hence will shift to using a general purpose timer that is more suitable for the needs of this (and future) labs. *Note: There are only a few differences between the two, and this manual will highlight those differences.* General purpose timers can be used in a variety of ways, and can tie directly to other peripherals. We'll use general purpose timers in this lab as a generic timer (similar to how SysTick would be used). There is also a Pulse Width Modulation (PWM) timer that is used for generating pulse sequences. We'll cover PWM in a later lab. Last, there is a watchdog timer - a timer that's reserved for helping the MCU recover from a bad situation. When a watchdog timer reaches the specified count value, it will automatically reset the MCU to hopefully recover from whatever error has caused it to get stuck. We'll talk more about these in another lab.

2.4.2 Configuring a General Purpose Timer

Most MCUs have several general purpose timers. Each can be configured differently to give several timing options in a design. In this MCU, there are six 32-bit timer blocks AND six 64-bit timer blocks that are divided into two halves (namely, TimerA and TimerB). This allows us to choose to use TimerA and TimerB as independent (split) timers, or we can combine them to give us larger timers. That gives us a total possibility of 24 different hardware timers that we have access to in our design. In the datasheet, 32-bit timers are referred to as GPTM (general purpose timer modules), and 64 bit timers are referred to as Wide General Purpose Timer Modules. The only difference (for now) that you should be concerned with is that *all* registers in this MCU are 32 bits wide. This means that wide timer modules will require TWO reads and TWO writes to get the information from a wide timer. For now, we'll just use a regular GPTM.

This MCU has an system clock frequency of 16Mhz by default (that's configurable up to 80Mhz). Much to our convenience, this MCU has a 1:1 relationship between system clock frequency and instruction clock frequency. This means our timer will count at a rate of 16Mhz by default. At 16Mhz, how much time passes between counts?

$$\frac{1}{16*10^6} = 62.5ns \tag{1}$$

If we have a 16 bit timer, how long will it take to count to the maximum value?

$$\frac{2^{16}-1}{16*10^6} = 4.1ms \tag{2}$$

A 32 bit timer?

$$\frac{2^{32}-1}{16*10^6} = 268.4s \tag{3}$$

A 64 bit timer?

$$\frac{2^{64}-1}{16*10^6} = 36559years \tag{4}$$

In addition to these numbers, there is a register that enables a prescale to these values; this adds to their durations by a configurable scaled amount. From page 760/761, we can see that a scaler can be applied to either TimerA or TimerB, and for the 16/32 bit timer a prescaler of up to 8 bits (256) can be applied. For the Wide Timers, the prescaler supports up to 16 bits. This gives us incredible control of the duration of our timers!

There are many different ways to configure these timers - most of which are out of scope of this lab. For now, let's walk through a **periodic** (a timer that automatically restarts when lapsed) configuration as outlined on page 722.

Here are some notes that may help you understand the instructions to configure periodic timer mode as outlined on page 722:

- Periodic means automatically restart, one-shot requires your code to restart the timer after it has lapsed. Choose what is most suitable to your application.
- It's up to you if you prefer a timer to count up or down. If counting up, you'll likely want to start counting from zero. If counting down, you'll likely start counting from some calculated value..
- Interrupts will be discussed in the next lab, and should be disabled for now.
- The last step refers to polling the RIS register. This register will tell you when a timer has lapsed. You can also view the value in the timer register directly by looking at the TAV and TAR registers directly. A helpful hint is that you can reset the timer value in your code as well; simply write a value to the TAV register to do so.
- The list of configuration registers are listed on page 726, but most of them are not needed. Just follow the instructions on page 722.

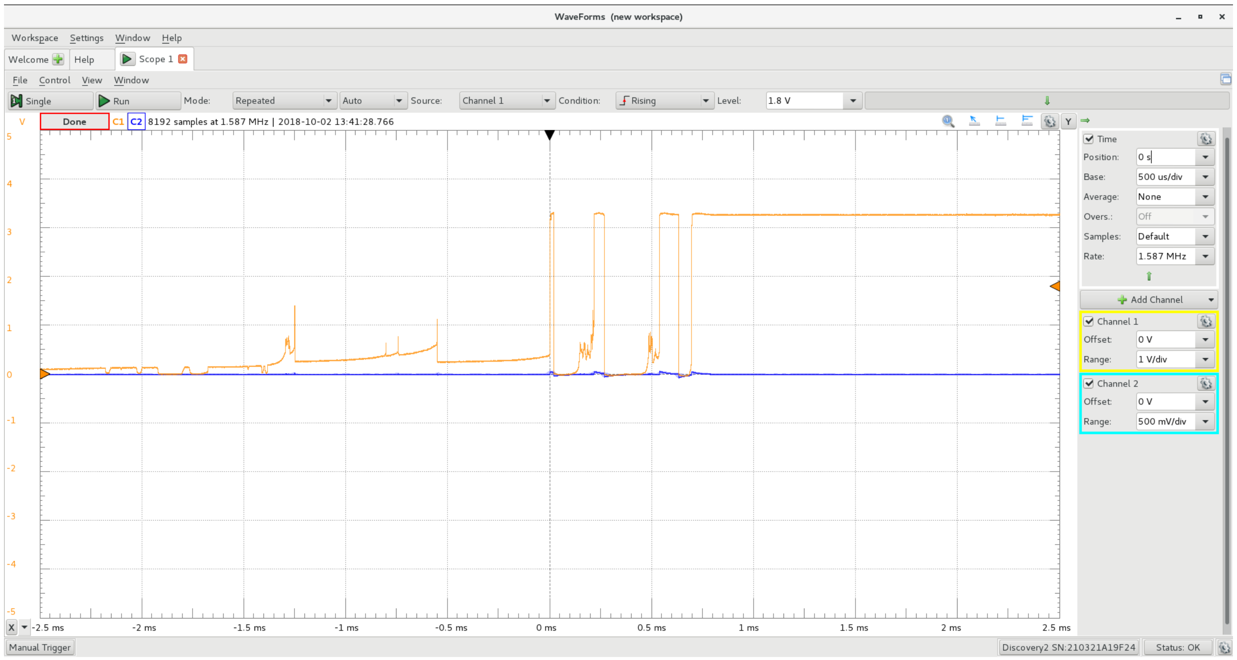
There are some differences between a general purpose timer and the system timer (SysTick). Namely, SysTick has fewer configuration options:

- SysTick can only be a 16 bit timer.
- SysTick can not have any scalers applied.
- SysTick can not be used to control/use with another peripheral directly (such as PWM, CCP, etc.)

2.5 Switch Bounce and Jitter

Converting our code from Lab 1 to non-blocking code will greatly improve what we can do with the microcontroller, but it also introduces some new problems.

Button presses (from any switch or button) are not ideal. Observe the waveform in Figure 3. This demonstrates how much jitter and bouncing these (unfiltered) mechanical switches have on release. The blocking solution from Lab 1 hid this problem from you because the blocking delays prevented button presses from occuring faster than 0.5 seconds.



WaveForms (new workspace)
Workspace Settings Window Help
Welcome Help Scope 1
File Control View Window
Single Run
Done C1 C2
V
5
4
3
2
1
0
-1
-2
-3
-4
-5
X -2.5 ms -2 ms -1.5 ms -1 ms -0.5 ms 0 ms 0.5 ms 1 ms 1.5 ms 2 ms 2.5 ms
Manual Trigger
Add Channel
Y

Mode: Repeated, Auto | Source: Channel 1 | Condition: Rising | Level: 1.8 V

8192 samples at 1.587 MHz | 2018-10-02 13:41:28.766

Time — Position: 0 s; Base: 500 us/div; Average: None; Overs.: Off; Samples: Default; Rate: 1.587 MHz

Channel 1 — Offset: 0 V; Range: 1 V/div

Channel 2 — Offset: 0 V; Range: 500 mV/div

Discovery2 SN:210321A19F24 | Status: OK

Figure 1: Waveform of a switch being released on the Launchpad.

With our newly found non-blocking solution to flashing an LED and checking the switch, the MCU will process the switch fast enough to detect each of the bounces as individual button presses! These bounces are not consistent, and are not always present. Similar problems can also exist on the press of the switch, and an undetermined amount of time for how long the user presses the switch makes using these buttons a real tricky problem. To avoid the detection of these bounces, we will need to add to our code 'button debouncing' where the code ignores the detection of these false presses. In particular, you will need to address three areas of problems:

1. Bounce/Jitter at the start of the press.
2. Holding the button down for an undertermined amount of time.
3. Bounce/Jitter at the release of the press.

There are -**many**- ways to address these problems. We can add hardware filtering and circuits to give us a clean single pulse, or we can apply a software fix to the issue. Adjusting the hardware is not an option in this case, so software it is! You should take some time to devise a solution to this problem, but if you are stuck here is a hint to one possible solution: *After the detection of a press (the first negative edge), we can safely ignore future button presses until the GPIO line remains high for at least 5ms (consecutively). You may find the non-blocking timers from Example 2.3 useful here.*.

3 Procedure

1. Create a new project called NSID_Lab_2. Refer to Lab 1 if you need help.
2. Copy your solution from Lab 1 into this new project.
3. Initialize Timer0 to count up in 32 bit mode. Set the upper bound of the timer to its maximum value (interval load), no interrupts, no prescale, and periodic mode. Follow the guide on Page 722 (but pay attention to adjust the values to match the requirements of this lab). To do this, you will need the following Symbol Definitions (or equivalents) . The register map is on pg. 726.
   (a) SYSCTL_RCGCTIMER_R (Page 338)
   (b) TIMER0_CTL_R
   (c) TIMER0_CFG_R
   (d) TIMER0_TAMR_R
   (e) TIMER0_IMR_R
   (f) TIMER0_TAILR_R
   (g) TIMER0_TAPS_R
   (h) TIMER0_TAV_R

   Make sure you read the register descriptions and what each bit does in each register!
4. Practice using the debugger. Read Section 2.1 if you haven't yet. Initialize and start the timer. View that the timer is working by using the debugger.
5. Adjust the GPIO initialization to include SW2. (You will need to unlock it first).
6. Test that SW2 is working by replacing the functionality of SW1 with SW2 (make SW2 change the LED color).
7. Adjust your blinking led solution to be a 'non-blocking' solution. Use the hardware timer to assist you.
8. Add switch debouncing code to make the switches work more reliably. See the appendix for additional help.

Exercise 3.1 (*Blinking LEDs and Polling SW1 and SW2*) $*$

1. Write a program that does the following (one program that does all of the following):
   (a) Flash the LED at a rate of **one** hertz *using a general purpose timer.*. (that's one blink per second).
   (b) Each time SW1 is pressed, change the color of the LED in the following order: (Red->Blue->Green->Repeat)
   (c) Make SW2 a Start/Stop blinking button. Each time it's pressed, start/stop blinking the LED (if not blinking, the LED should be ON all the time).
2. Make a PDF that has any information that will benefit the marker in regards to the coding question, and answers to the following questions:
   (a) Draw a flow chart of your solution. Make rectangles for 'Do something and continue' blocks and diamonds for 'Decision' blocks.
   (b) Briefly describe the difference **in your own words** between blocking and non-blocking code.
   (c) Include a screenshot of your debugging session showing the correct configuration of PORTF (GPIO_PORTF_DIR, DEN, PUR, AFSEL) through the Registers view.
   (d) If the system clock frequency was 80Mhz, how much time would pass between timer counts? How long for a 16-bit timer to lapse? 32-bit timer? 64-bit timer?
3. Go back to the start of this manual and verify that you have met the objectives of the lab.

4 Hand-in Instructions

Make a folder called 'Documents' inside of the folder that is your solution to Exercise 3.1 (that should be named NSID_Lab_2). Place your PDF with the answers to your questions in the Documents Folder. Zip the entire contents of the folder named NSID_Lab_2, and hand-in the zip file to canvas/blackboard before the due date. The due date is one week from the day of the lab (at midnight). See the note on the first page about working with a partner.

5 Appendix A: Debouncing Help

Button Debouncing can be a difficult thing to implement. Every switch will have a different profile and characteristic, so there is not a single debouncing technique that works for every situation. As such, this information is tailored to the launchpads specifically.

From the first lab, you should have a simple blocking solution that could look like this:

```
int color;
while(1)
{
  if(sw1) //keep track of what color we are on
    if(color == red)
      color = green;
    else
      color = red

  toggle_led(color); //toggle the led based on the color
  delay(500ms);
}
```

The problem with the solution is that we are only checking the switch every 500ms. That's not fast enough.

We can make this better by reducing the blocking delay duration:

```
int color = red;
int count = 0;
while(1)
{
  if(sw1) //keep track of what color we are on
    if(color == red)
      color = green;
    else
      color = red

  if(count == 100) // 5ms * 100 = 500ms
  {
    toggle_led(color); //toggle the led based on the color
    delay(5ms); //wait some time
    count = 0;
  }
  count++; //increments every iteration of the while loop.
}
```

Although this is better, we can be quicker to respond to button presses by eliminating the delay entirely and implementing a software timer.

```
int color = red;
int count = 0;
while(1)
{
  if(sw1) //keep track of what color we are on
    if(color == red)
      color = green;
    else
      color = red
  if(count == 50000) // estimating that every 50000 counts is close to 500ms.
  {
    toggle_led(color); //toggle the led based on the color
    count = 0;
  }
  count++;
}
```

This solution is a non-blocking solution, but is difficult to maintain.. as we add more code to the while(1) loop, the timing of our software timer (count) changes. Thus, we need to constantly be updating the count value to keep it close to 500ms. **This solution also has some issues with button bouncing and pressing and holding** the button will cause our color to change many times per press.

To get a more accurate timer, we need to convert this solution to using a hardware timer instead:

```
int color = red;
while(1)
{
  if(sw1) //keep track of what color we are on
    if(color == red)
  color = green;
    else
  color = red

  if(TIMER0_TAV > 8000000) // 500ms has elapsed.
  {
    toggle_led(color); //toggle the led based on the color
    TIMER0_TAV = 0; //reset the timer
  }
}
```

At this point no button presses will be missed, but **we still have the issues of button bouncing and pressing and holding the button**.

Looking at a standard button press, we can see that we need to ignore presses for a while after accepting a button press.

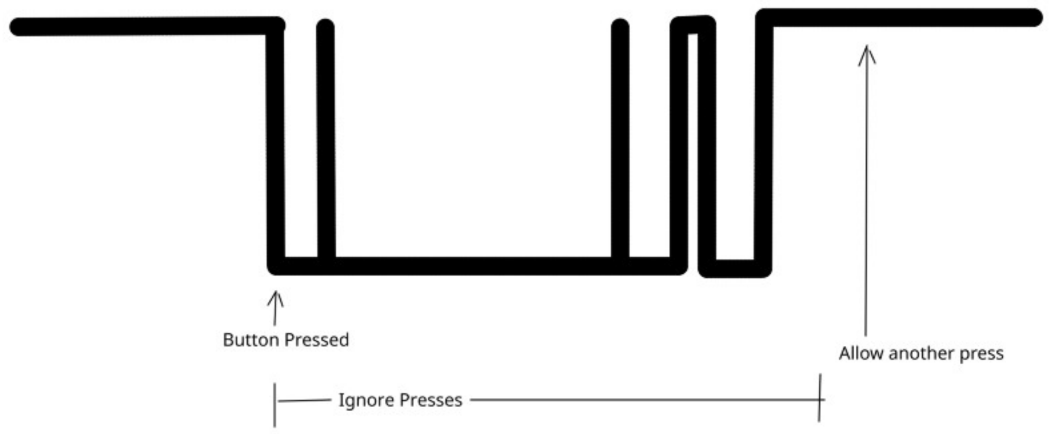


Figure 2: Button Debouncing Technique

The easiest way to implement debouncing (and deal with the press and hold at the same time) is to only allow another button press to be accepted **if the switch input has been released for at least 5 ms** after accepting a valid button press. The value 5ms is found in the datasheet of the switch as the worst-case bounce time.

To implement this, we can use the hardware timer as well:

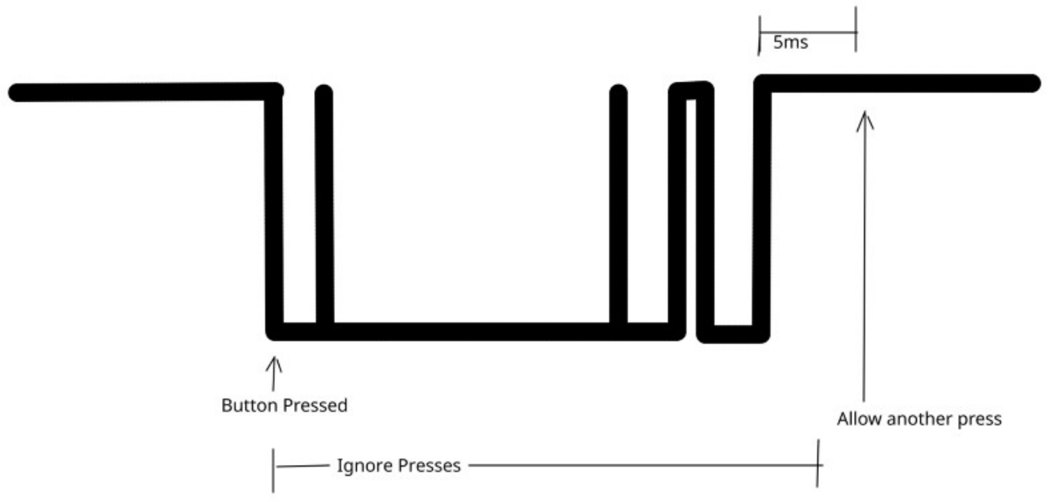


Figure 3: Button Debouncing Technique

```
int OK_For_SW1 = 1; //flag to indicate if it is ok to accept SW1 press
while (1)
{
  //Logic to accept a button press
  if (SW1 && OK_For_SW1)
  {
    OK_For_SW1 = 0; //Need to ignore presses for a while.
    TIMER0_TAV = 0; //reset debounce timer
    Do_Stuff(); //set flags to let other actions and events happen (such as change color of led)
  }

  //Logic to ignore presses
  if (SW1 && (!OK_For_SW1))
  {
    TIMER0_TAV = 0; //Keep resetting the debounce timer for as long as the button is held.
  }

  //Logic to know when it is OK to accept another button press again
  if ((!OK_For_SW1) && (TIMER0_TAV > 5ms))
  {
    OK_For_SW1 = 1; //reset flag
  }

  //add code here for other actions and events
}
```

Notice how the entire solution ONLY uses 'if' statements. **There are no blocking delays!**