Lab 4. Minimally Intrusive Debugging Methods

[Preparation](#_a68rv81q7zua)

[Purpose](#_boovat2ouzu5)

[System Requirements](#_tq09y9mq5cfa)

[Procedure](#_7rgpw2rhyl63)

[Part a - Write SysTick Initialization](#_4zg4aibpzc9r)

[Part b - Write Dump](#_lxm5tbew13xg)

[Part c - Estimate Intrusiveness](#_tlxij16d2uxr)

[Part d - Implement Heartbeat](#_qzcgxj72fx5n)

[Part e - Debug](#_jcaa45kr9jac)

[Part f - Capture Timing](#_2j4j2d38dp03)

[Demonstration](#_8ij66g2uvq6e)

[Deliverables](#_94qguhpy7g3)

[Debugger output → file](#_3wtaxahias7h)

[FAQ](#_4tklqfrx9lm4)

## Preparation

Read Sections 4.3, 4.4, 5.1, 5.2, 5.5, 5.7, 6.1, 6.2, and 6.9

Download, unzip, open, compile, and run the project

**ECE3436\_Lab\_4** Starter project from BlackBoard

These videos <https://www.youtube.com/playlist?list=PLyg2vmIzGxXHYEHQrxNxGcRg6vCTB20Ud> relate to Lab 4.

## Purpose

The purpose of this lab is to learn minimally intrusive debugging skills. When visualizing software running in real-time on an actual microcomputer, it is important to use minimally intrusive debugging tools. We call a debugging instrument minimally intrusive when the time it takes to collect and store the information is short compared to the time between when information is collected. In particular, you will learn to use both a dump and a heartbeat.

The first objective of this lab is to develop an instrument called a dump, which does not depend on the availability of a debugger. A dump allows you to capture strategic information that will be viewed at a later time. Many programmers use the printf statement to display useful information as the programming is being executed. On an embedded system we do not have the time or facilities to use printf. Fortunately, we can use a dump in most debugging situations for which a printf is desired. Software dumps are an effective technique when debugging real-time software on an actual microcomputer.

The second useful debugging technique you will learn is called a heartbeat. A heartbeat is a visual means to see that your software is still running. The debugging techniques in this lab use both hardware and software and are appropriate for the real TM4C123. Software skills you will learn include indexed addressing, array data structures, the PLL, the SysTick timer, and subroutines.

## System Requirements

In this lab, you will design, implement, test, and employ software debugging instruments to experimentally verify the correct operation of your Lab 2-3 system. You will only verify the variable duty-cycle feature of Lab2-3 and not the “breathing” feature. The other change in requirements is, what happens to the LED when the button is pressed and before it is released. We require you to operate the LED at the current duty-cycle (see Figure 4.2 below) and change it to the new duty-cycle when the button is released.

* You will activate the **SysTick** timer (call **SysTick\_Init** that you implement in **SysTick.s**), which will make the 24-bit counter, **NVIC\_ST\_CURRENT\_R**, decrement every 12.5 ns. We will use this counter to measure time differences up to 210 ms. To measure the current time, you simply read the 24-bit **NVIC\_ST\_CURRENT\_R** value.
* The LED toggles at 8 Hz and a varying duty-cycle. Repeat the functionality from Lab2-3 but now we want you to insert debugging instruments which gather data (state and timing) to verify that the system is functioning as expected.
* Hardware connections (External: One button and one LED)
  + PE1 is Button input (1 means pressed, 0 means not pressed)
  + PE0 is LED output (1 activates external LED on protoboard)
  + PF2 is Blue LED on Launchpad used as a Heartbeat
* Instrumentation data to be gathered is as follows: On Button(PE1) press collect one state and time entry; On Button release, collect 7 state and time entries on each change in state of the LED(PE0): An entry is one 8-bit entry in the Data Buffer and one 32-bit entry in the Time Buffer:
  + The Data Buffer entry (byte) content has:
    - Lower nibble is state of LED (PE0)
    - Higher nibble is state of Button (PE1)
  + The Time Buffer entry (32-bit) has:
    - 24-bit value of the SysTick's Current register (**NVIC\_ST\_CURRENT\_R)**
* Note: The size of both buffers is 50 entries. Once you fill these entries you should stop collecting data
* The Heartbeat is an indicator of the running of the program. On each iteration of the main loop of your program toggle the LED to indicate that your code(system) is live (not stuck or dead).

## Procedure

The basic approach to Lab 4 through Lab 9 will be to first develop and test the system using simulation mode. After debugging instruments themselves are tested, you will collect measurements on the real TM4C123.

### Part a - Write SysTick Initialization

To use the SysTick countdown timer it has to be initialized first. The SysTick\_Init routine implemented inside SysTick.s is responsible of this task. You have to the write the steps involved in SysTick initialization in this file. You will setup SysTick to run continuously with no interrupts. You will use the current value of the SysTick to mark the time (in the **TimeBuffer**) at which an entry is made in the debugging instrument **DataBuffer** array.

### Part b - Write Dump

When your main program calls **TExaS\_Init**, this subroutine will activate the **PLL** making bus clock 80 MHz. So all time calculations use a 12.5ns clock cycle. Write two debugging subroutines, **Debug\_Init** and **Debug\_Capture**, that implement a dump instrument. They will together save both input/output, and timing data. If we saved just the input/output data, then the dump would be called *functional debugging* because we would capture input/output data of the system without timing information. However, you will save both the input/output data and the time, so the dump would be classified as *performance debugging*.

You will define an array capable of storing 50 entries of Port E measurements(8-bit), and an array capable of storing 50 entries of time measurements(32-bit).

The first subroutine (**Debug\_Init**) initializes your dump instrument. The initialization should activate the SysTick timer, place 0xFFFFFFFF into the two arrays to signify that no data has been saved yet, and initialize pointers and/or counters as needed. The second subroutine (**Debug\_Capture**) that saves one data-point (**PE1** input data, and **PE0** output data) in the first array and the **NVIC\_ST\_CURRENT\_R** in the second array. Since there are only two bits to save in the first array, pack the information into one value for ease of visualization when displayed in hexadecimal. Put the PE1 value into bit 4 and the PE0 value into bit 0. The table below illustrates how this makes the data easier to visualize after a dump is taken.

|  |  |  |
| --- | --- | --- |
| **Input (PE1)** | **Output (PE0)** | **save data** |
| 0 | 0 | 000**0**,000**0**2, or 0x00000000 |
| 0 | 1 | 000**0**,000**1**2, or 0x00000001 |
| 1 | 0 | 000**1**,000**0**2, or 0x00000010 |
| 1 | 1 | 000**1**,000**1**2, or 0x00000011 |

Place a call to **Debug\_Init** at the beginning of the system, and a call to **Debug\_Capture** at the start of each execution of the outer loop. The basic steps involved in designing the data structures for a pointer implementation of this debugging instrument are as follows.

1. Allocate **DataBuffer** in RAM (to store 50 entries of state data)
2. Allocate **TimeBuffer** in RAM (to store 50 seconds of timer data)
3. Allocate two pointers (**DataPt**, **TimePt**), one for each array, pointing to the place to save the next data

The basic steps involved in designing **Debug\_Init** are as follows, assuming a pointer scheme

1. Set all entries of the first **DataBuffer** to 0xFF (meaning no state saved yet)
2. Set all entries of the second **TimeBuffer** to 0xFFFFFFFF (meaning no timing saved yet)
3. Initialize the two pointers to the beginning of each buffer
4. Activate the SysTick timer (call **SysTick\_Init**)

The basic steps involved in designing **Debug\_Capture** are as follows, again assuming a pointer scheme

1. Save any registers needed
2. Return immediately if the buffers are full (**NEntries** is 50)
3. Read Port E and the SysTick timer (**NVIC\_ST\_CURRENT\_R**)
4. Mask capturing just bits 1,0 of the Port E data
5. Shift the Port E data bit 1 into bit 4 position, and leave bit 0 into bit 0 position
6. Dump this port information into **DataBuffer** using the pointer **DataPt**
7. Increment **DataPt** to next address
8. Dump time into **TimeBuffer** using the pointer **TimePt**
9. Increment **TimePt** to next address
10. Update **NEntries**
11. Restore any registers saved and return

For regular functions we are free to use R0, R1, R2, R3, and R12 without preserving them. However, for debugging instruments, we should preserve all registers, so that the original program is not affected by the execution of the debugging instruments. The temporary variables may be implemented in registers. However, the buffers and the pointers should be allocated in RAM. You can observe the debugging arrays using a Memory window. Look in the map file to find the addresses of the buffers.

### Part c - Estimate Intrusiveness

One simple way to estimate the execution speed of your debugging instruments is to assume each instruction requires about 2 cycles. By counting instructions and multiplying by two, you can estimate the number of cycles required to execute your **Debug\_Capture** subroutine. Assuming the 12.5 ns bus cycle time, convert the number of cycles to time. Next, estimate the time between calls to **Debug\_Capture**. Calculate the percentage overhead required to run the debugging instrument (100 times execution time (x2) divided by time for one cycle of 8Hz). The execution time is multiplied by 2 because in each 8Hz cycle we call the instrument twice. This percentage will be a quantitative measure of the intrusiveness of your debugging instrument. Add comments that include these estimations and calculations to your program.

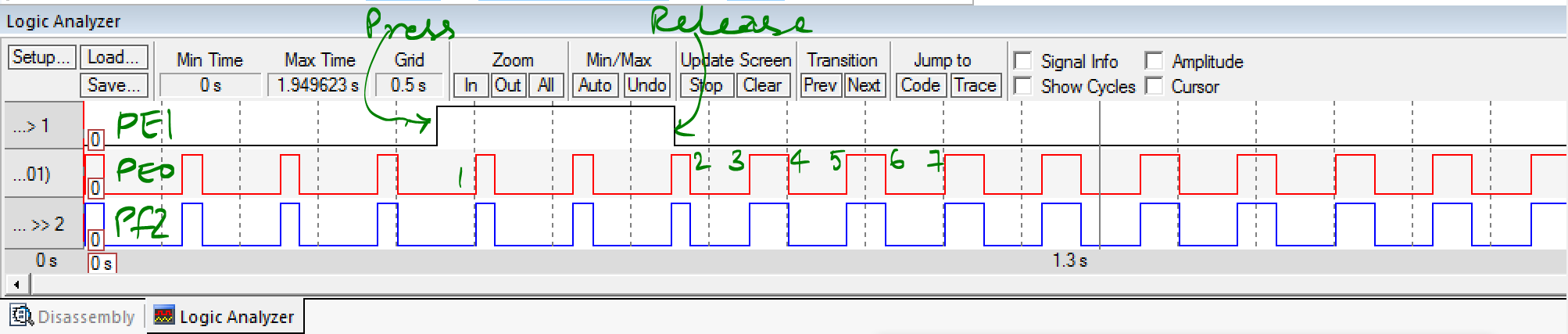
### Part d - Implement Heartbeat

Write debugging software and add **PF2** as an output to the Lab 2-3 system so that this onboard Blue LED always flashes while your program is running. In particular, initialize the direction register so **PF2** is an output, and add code that toggles the LED each time through the loop. A heartbeat of this type will be added to all software written for Labs 4, 6, 7, 8, and 9 in this class. Lab 5 is omitted as we will use PF2 as part of the system output. A heartbeat is a quick and convenient way to see if your program is still running.

When you get to EE445L you can implement multiple heartbeats at various strategic places in the software and that toggle much faster than the eye can see. In these situations, you will use a logic analyzer or oscilloscope to visualize many high-speed digital signals all at once. However, in EE319K there will be one heartbeat on **PF2**, and the heartbeat must occur slow enough to be seen with the eye.

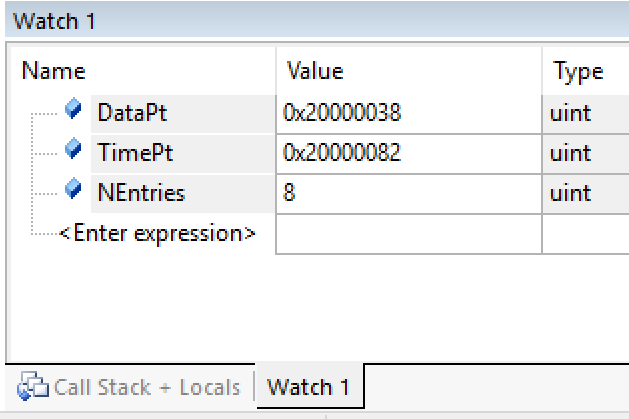
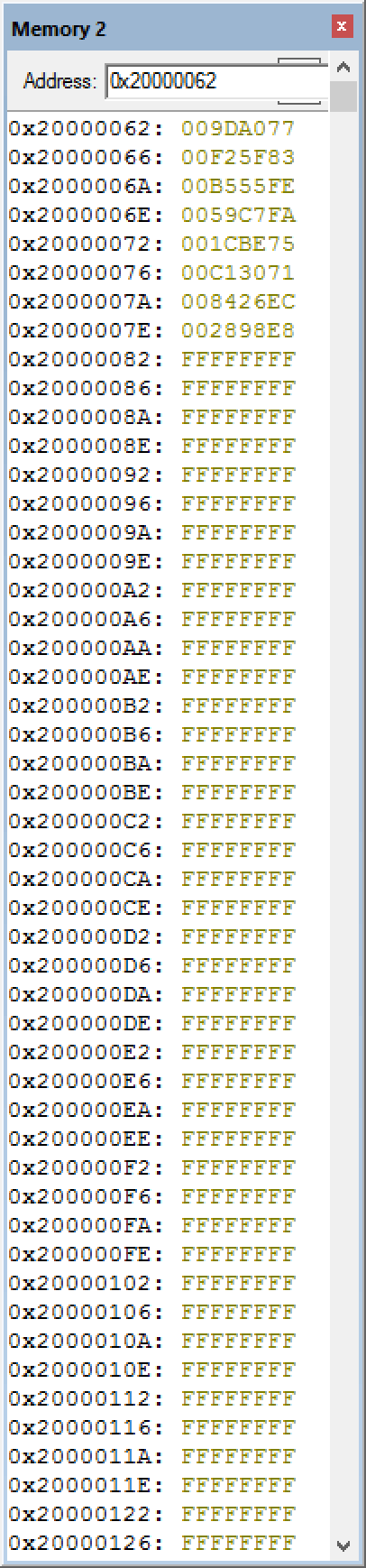
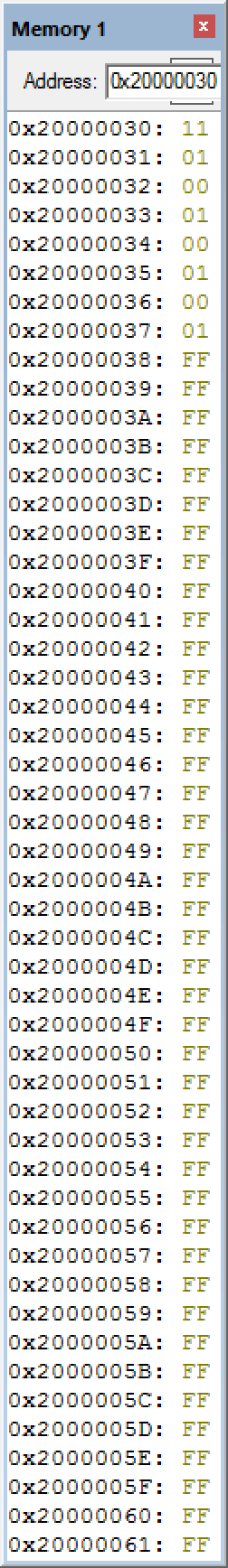
### Part e - Debug

Debug your combined hardware/software system first with the simulator, then on the actual TM4C123 board. You can find the address of the buffers by looking in the map file. You can dump the memory to a file using the command (the data is formatted in a little-endian hexadecimal format) ‘**SAVE data.txt 0x20000000 , 0x20000190’** replacing the values 0x20000000 and 0x20000190 with the start and end addresses of your arrays .

****

*Figure 4.2. Simulation output showing the input on PE1, output on PE0 and heartbeat on PF2.*

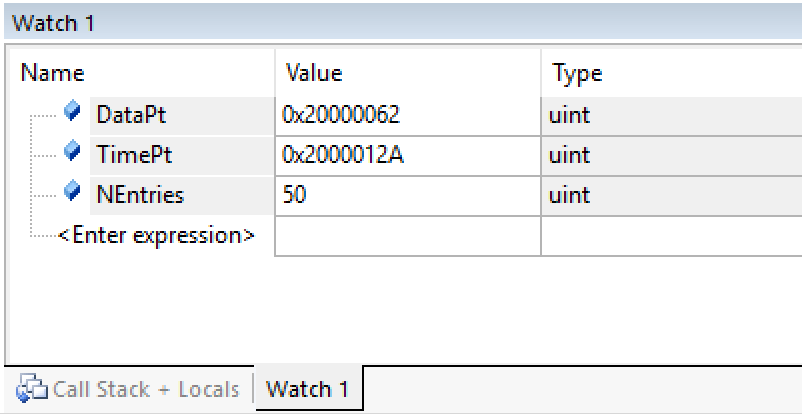
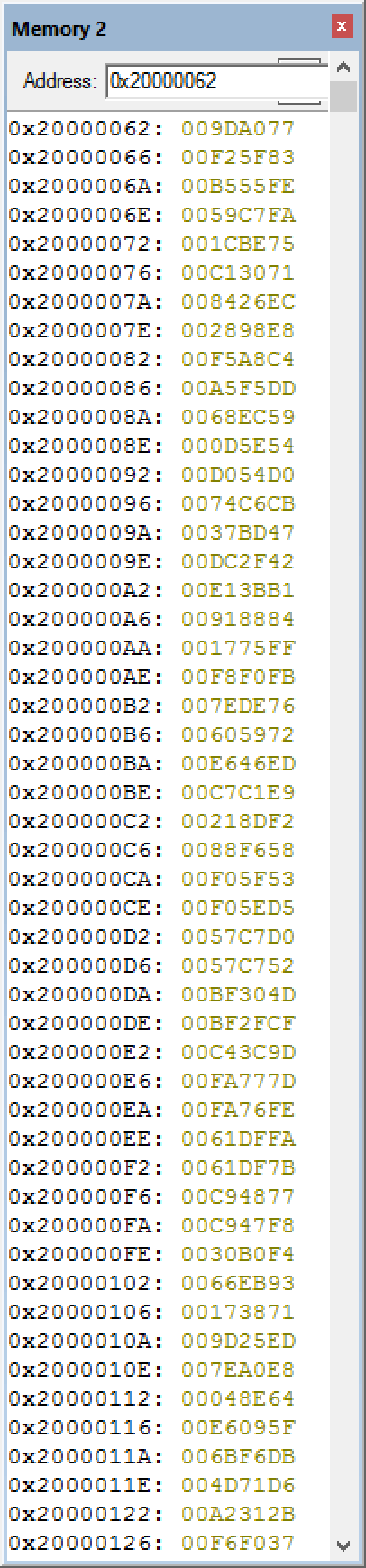
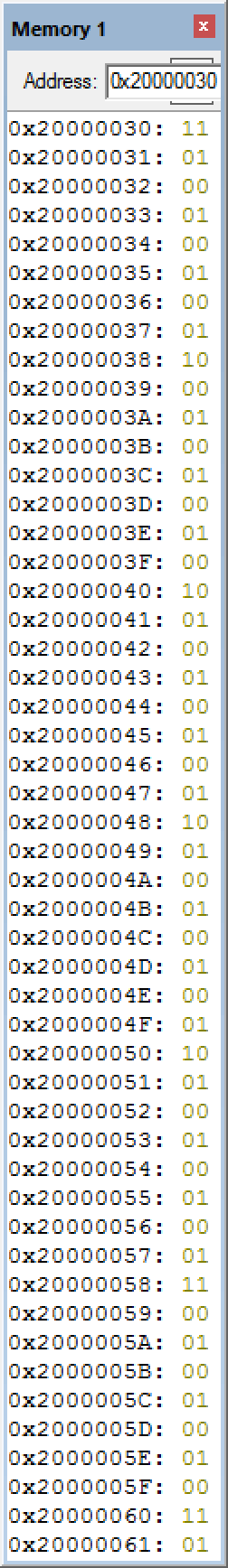
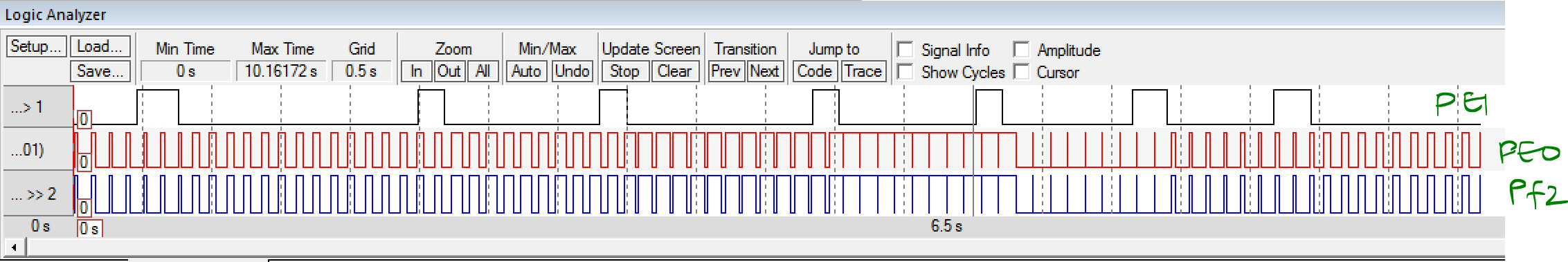
Note that the ten data points entered into the DataBuffer are numbered in the Figure. The figure (4.3) below shows the memory window for both the DataBuffer and TimeBuffer



*Figure 4.3. State and Timings are observed in the respective memory windows and on the real board showing results of the dump after the first 8 entries were made. The DataPt and TimePt reflect the position in the respective buffers*

### Part f - Capture Timing

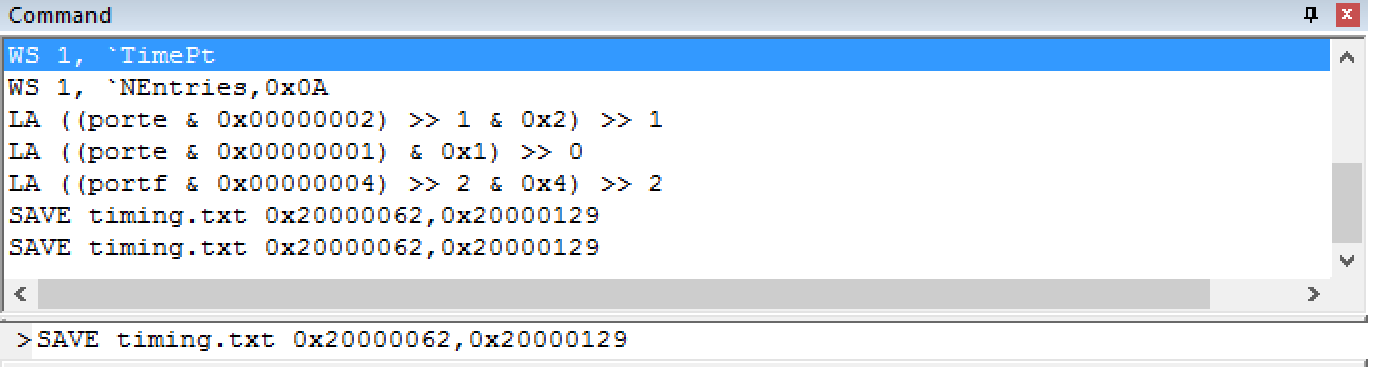
Run your debugging instrument capturing the sequence of PE1 inputs and PE0 outputs as you touch, then release the switch. You will collect performance data on the system as described in the System requirements above.



*Figure 4.4. State and Timings are observed in the respective memory windows in simulation and on the real board showing results of the dump after all 50 entries were made. The DataPt and TimePt reflect the position in the respective buffers*

## Debugger output → file

How to transfer data from debugger to a computer file

1. Run your system so data is collected in memory, assume interesting stuff is from 0x20000062 to 0x20000126; Note the last entry ends at 0x20000129
2. Type **SAVE timing.txt 0x20000062 , 0x20000129** in the command window after the prompt (“>”), type enter
3. Open the **timing.txt** file in NotePad, it is an standard format called the [Intel Hex](http://www.keil.com/support/docs/1584/) format(we encourage you to read the format documentation). Here is how it looks:

:020000042000DA

:0E006200**77A09D00835FF200FE55B500FAC7**3F

:10007000**590075BE1C007130C100EC268400E898**60

:10008000**2800C4A8F500DDF5A50059EC6800545E**11

:10009000**0D00D054D000CBC6740047BD3700422F**AE

:1000A000**DC00B13BE10084889100FF751700FBF0**94

:1000B000**F80076DE7E0072596000ED46E600E9C1**88

:1000C000**C700F28D210058F68800535FF000D55E**1E

:1000D000**F000D0C7570052C757004D30BF00CF2F**98

:1000E000**BF009D3CC4007D77FA00FE76FA00FADF**7F

:1000F000**61007BDF61007748C900F847C900F4B0**B0

:10010000**300093EB660071381700ED259D00E8A0**E4

:10011000**7E00648E04005F09E600DBF66B00D671**9A

:0A012000**4D002B31A20037F0F600**6D

:00000001FF

1. To get to the data dumped from memory, strip off the first 9 characters of every line, and the last two characters of every line. Ignore the first and last lines, leaving the data shown above in bold. Each two characters is a byte in hex. 32-bit and 16-bit data are of course little endian. The first and last record are marked in red for your reference. Alternatively, you can cut and paste the saved text from the **timing.txt** file for automatic analysis into the excel file (Calculation.xlsx). The excel file should be self-explanatory.

## Demonstration

**(both partners must be present, and demonstration grades for partners may be different)**

You will show the TA your program operation on the actual TM4C123 board. The TA may look at your data and expect you to understand how the data was collected and what the data means. Also be prepared to explain how your software works and to discuss other ways the problem could have been solved. Questions that may be asked may include:

* What does **Texas\_Init** do?
* The TA will pick an instruction in your program and ask how much time does it take that instruction to execute in μsec. Does it always take same amount of time to execute?
* You will be asked to create a breakpoint, and add the port pin to the simulated logic analyzer.
* Is **Debug\_Capture** minimally intrusive or non-intrusive?
* What do you mean by intrusiveness?
* Is your code “friendly”?
* How do you define masking?
* How do you set/clear one bit in without affecting other bits?
* What is the difference between the **B**, **BL** and **BX** instructions?
* How do you initialize the SysTick?
* You should understand every step of the function **SysTick\_Init**.
* How do you change the rate at which SysTick counts?
* Describe three ways to measure the time a software function takes to execute?
* How do you calculate the sizes of the port data and the timestamp data?
* If you used 32-bit data for **DataBuffer** what would be the advantages over 8-bit data?
* Could you have stored the time-stamp data in 8-bit, 16-bit, or 24-bit arrays?
* Why does the pointer to the time-stamp array need to be incremented by four, if you want to point to the next element in the array?
* How do you allocate global variables?
* Consider the four possible data values that could be stored into the **DataBuffer** : 0x00 (meaning In=0, Out=0), 0x01 (meaning In=0, Out=1), 0x10 (meaning In=1, Out=0), and 0x11 (meaning In=1, Out=1). Are all of these values likely to occur, what do they indicate?

## Deliverables

1. **Printout of your code**

# **FAQ**

The list of FAQ below are populated from Piazza over the semesters (thanks to the contributions of all past TAs and students). More questions may be posted so please check back regularly.

1. **Should our arrays be located somewhere specific? How large is it supposed to be, and how can we make sure nothing else writes into that address?**

They are located in RAM but the actual locations do not matter. The declarations look like:  
 DataBuffer SPACE 50  
 TimeBuffer SPACE 4\*50  
 DataPt SPACE 4  
 TimePt SPACE 4

You have to make sure that you access them properly to avoid corrupting their contents

1. **I'm getting the following error warning when I try to build my code:**

**Error: L6238E: main.o(.text) contains invalid call from '~PRES8 (The user did not require code to preserve 8-byte alignment of 8-byte data objects)' function to 'REQ8 (Code was permitted to depend on the 8-byte alignment of 8-byte data items)' function TExaS\_Init.**

Are you pushing and popping an even number of registers in your program? AAPCS requires you to push registers in multiples of 2. Also, be sure to always balance the stack, meaning have the same number of pops as pushes.

An alternative to pushing and popping an even number of registers is to add "PRESERVE8" above the AREA command at the beginning of your program. You can add this to your assembly files. By doing this, you are basically lying to the compiler that you are "promising to actually push and pop an even number of registers."

1. **So I'm getting a percentage overhead of 0.0011%. Does that seem reasonable?**

The overhead will be dominated by how much time you delay for in your loop. 125ms is a *ton* of time when the clock ticks at 80MHz. Further, we did ask you to implement a minimally intrusive debugging instrument. 0.0011% sounds minimally intrusive to me.

1. **My program originally worked as planned, but it did not store the information at the proper locations, so I just changed the pointer increment from 1 to 4. While it compiles properly, the program now does not run in debug mode, displaying the following error:**

**Error 65: access violation at 0x20008000 : no "write" permission**

That address, 0x20008000, is one byte after the end of ram, which has a range of 0x2000.0000 to 0x2000.7FFF and is therefore 32kB. Your program is trying to store to memory that does not exist.This is a problem with how you wrote some of your code. In particular this looks like a missing bounds check. Normal debugging (stepping, breakpoints) should be sufficient to find the problem

1. **Inside SysTick.s do we actually have to write the code for the SysTick\_Init subroutine?**

Yes. Also, you have to make the linkage between the SysTick\_Init routine implemented inside SysTick.s and the caller (main.s) using IMPORT and EXPORT statements.

1. **We have the values and they look somewhat like what is at the end of the lab manual, but we don't know how to interpret the data**

Remember that your dump writes both a port E capture, and a timing (SysTick) capture every time you call it. This implies that the arrays are paired, where each Port E word pairs with a SysTick value. once you match these up, take the difference between adjacent SysTick values to generate the time delta. Simply convert this delta into seconds (remember that the clock is 80 MHz, which implies that 1 cycle = 12.5 ns) and take the average.

1. **What is the advantage of having an 8 bit data buffer? is it just so that it would save to memory quicker? and a 32 bit simply holds more data but saves slower?**

8 bit/1 byte data buffers take less space, and hold less data.

32 bit/4 byte data buffers take more space and hold more data.

There is no difference in speed between them

1. **After I debug my code this error appears: Error: Could not load file 'C:\Keil\EE319KwareSpring2016\Lab4\_EE319K\Lab4.axf'. Debugger aborted !**

That error message is telling you that you do not have an executable file (.axf) and it therefore can not continue.

Reasons that you don't have an executable file may include:

* You did not compile/build
* Last time you compiled/built the executable was not created due to errors

1. **0. What is the point of the lab?**This is a functional debugging lab. It is meant to involve collection of data/timing and analyze it offline to show the functioning of your software. As long as you collect data and are able to make intelligent observations by looking at the data you have met the intent of the lab.
2. **How many points to capture and when?**  
   Here is what the lab manual says:   
   “On Button release, collect 7 state and time entries on each change in state of the LED(PE0)”  
   seems very clear that the collection is on state change of LED which means on to off or off to on. Yes, the 0% and 100% duty-cycle scenarios are not clear depending on how you implement these situations. If you handled these two scenarios as exceptions then your captured data will be complicated. If you handled them as just any other duty-cycle then you will see short spikes where the state of the LED still changes, albeit for a short time and the captured data will still make perfect sense.   
   On a related note, 8 entries per button press (duty-cycle change) will give you enough entries to see all duty-cycles operating.
3. **How should the heartbeat work?**  
   Ideally it should be an indicator of the "liveness" of your system. The simplest way to implement it is to put it at the beginning or end of the your outer while loop. Yes, given the nature of this lab it will track the duty-cycle and exhibit weird behavior at the 0 and 100% duty cycle. This is a problem for the current lab but will not be the case in future labs.
4. **What is the excel file for?**  
   The excel file is given to you to analyze the timing dump to see if your system is meeting the requirements. You could do the analysis manually if you choose, to by understanding what is being dumped and extracting the relevant data yourself. To use the excel file simply cut-and-paste the contents of the timing.txt that keil produces using the SAVE command (No need to strip off any characters).