

# Lab 4. Minimally Intrusive Debugging Methods

[Preparation](#)

[Purpose](#)

[System Requirements](#)

[Procedure](#)

[Part a - Write Dump Routine](#)

[Part b - Write Heartbeat](#)

[Part c - Estimate Intrusiveness](#)

[Part d - Implement Heartbeat](#)

[Part e - Debug](#)

[Part f - Capture Timing](#)

[Demonstration](#)

[Deliverables](#)

[Debugger output → file](#)

## 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

**SysTick\_4C123asm.zip**

PLL and SysTick functions

**Lab4\_EE319K\_asm**

Starter project with links to grading engine

These videos <https://www.youtube.com/playlist?list=PLyg2vmlzGxXHYEHQrxNxGcRg6vCTB20Ud> 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 3 system.

- You will record the Port E value (including both the input and output signals) and the time during each execution through the outer loop of the main program of Lab 3 as your system runs in real time.
- You will activate the **PLL** (by calling the **TEXaS\_Init** function, which in turn will execute Program 4.6 of the book) to make the microcontroller run at 80 MHz.
- You will activate the **SysTick** timer (call **SysTick\_Init** shown in Program 4.7 of the book), 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 335.5 ms. To measure the current time, you simply read the 24-bit **NVIC\_ST\_CURRENT\_R** value.
- The dump should store Port E and **NVIC\_ST\_CURRENT\_R** data into arrays while the system is running in real-time, and allow the information to be viewed at a later time.
- You will toggle an LED once each time through the loop to create a heartbeat.

## Procedure

The basic approach to Lab 3 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. As always, the **Grading Controls** are for the MOOC and are not used in EE319K this semester.

This lab add onto the code you have written for lab 3.

All new code written for this lab must be in assembly.

## Part a - Write Dump

When your main program calls **TEXaS\_Init**, this subroutine will activate the **PLL** making bus clock 80 MHz. Adjust the delay function so it delays 62 ms. 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 about 3 seconds worth of Port E measurements, and an array capable of storing about 3 seconds worth of time measurements. For example, if the outer loop of your Lab 3 executes in about 62 ms, then the loop will be executed about 50 times in 3 seconds (3000/62), so the array sizes will need to be 50 elements each. You may use either pointers or counters to store data in the arrays.

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	0000,0000 <sub>2</sub> , or 0x00000000
0	1	0000,0001 <sub>2</sub> , or 0x00000001
1	0	0001,0000 <sub>2</sub> , or 0x00000010
1	1	0001,0001 <sub>2</sub> , 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 3 seconds of input/output data)
2. Allocate **TimeBuffer** in RAM (to store 3 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 buffer to 0xFFFFFFFF (meaning no data yet saved)
2. Set all entries of the second buffer to 0xFFFFFFFF (meaning no data yet saved)
3. Initialize the two pointers to the beginning of each buffer
4. Activate the SysTick timer (call **SysTick\_Init** shown in Program 4.7 of the book)

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 (pointer past the end of the buffer)
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 0 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. Restore any registered 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. After you have debugged your code in simulation mode, capture a screenshot showing the results as the switch is pressed then not pressed. The dumped data should start with some 0x01 data, next it should oscillate between 0x10, 0x11 as the switch is pressed, then return to 0x01 when the switch is released.

## Part b - 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 divided by time between calls, in percent). 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 c - Implement Heartbeat

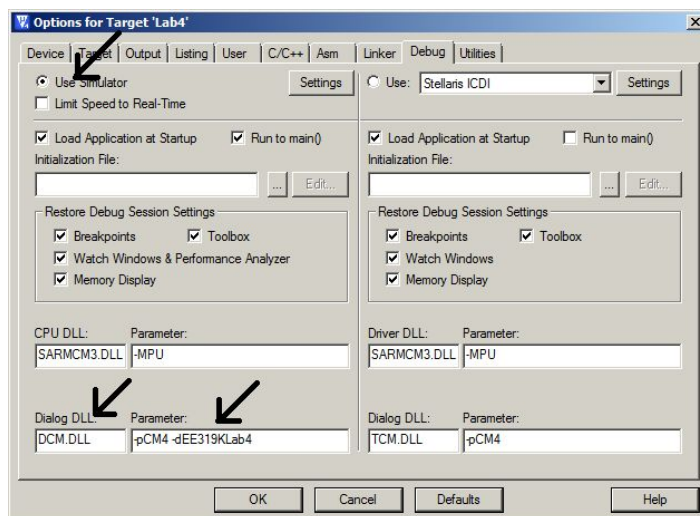
Write debugging software and add **PF2** as an output to the Lab 3 system so that this onboard LED always flashes while your program is running. In particular, initialize the direction register so **PF2** is an output, and add code that toggles the LED each time through the loop. A heartbeat of this 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 d - Debug

Debug your combined hardware/software system first with the simulator, then on the actual TM4C123 board. Figure 4.1 shows how to configure the Lab 4 simulator. Figure 4.2 is screenshot of Lab 4 in simulation mode. 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 .



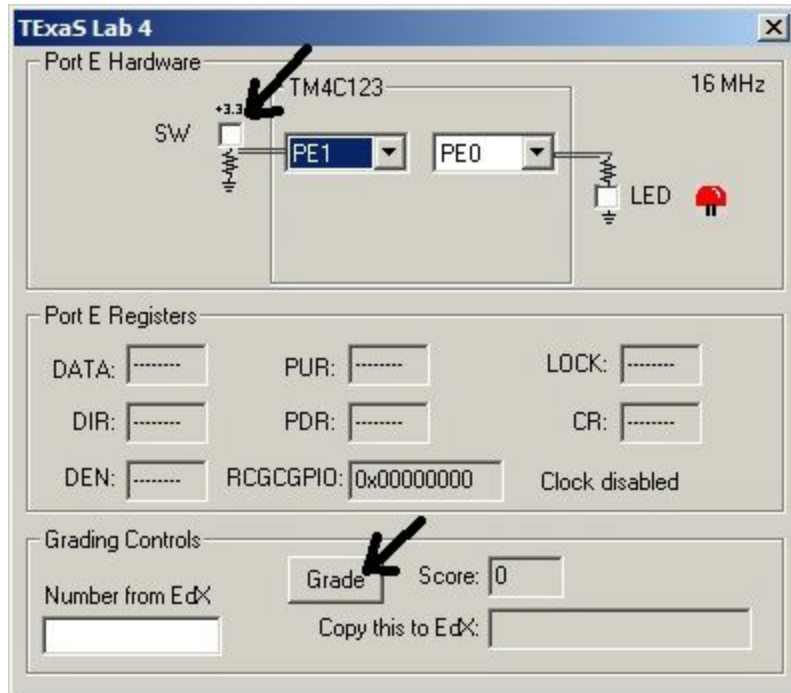


Figure 4.1. Using TExaS to debug your software in simulation mode (DCM.DLL -pCM4 -dEE319KLab4).

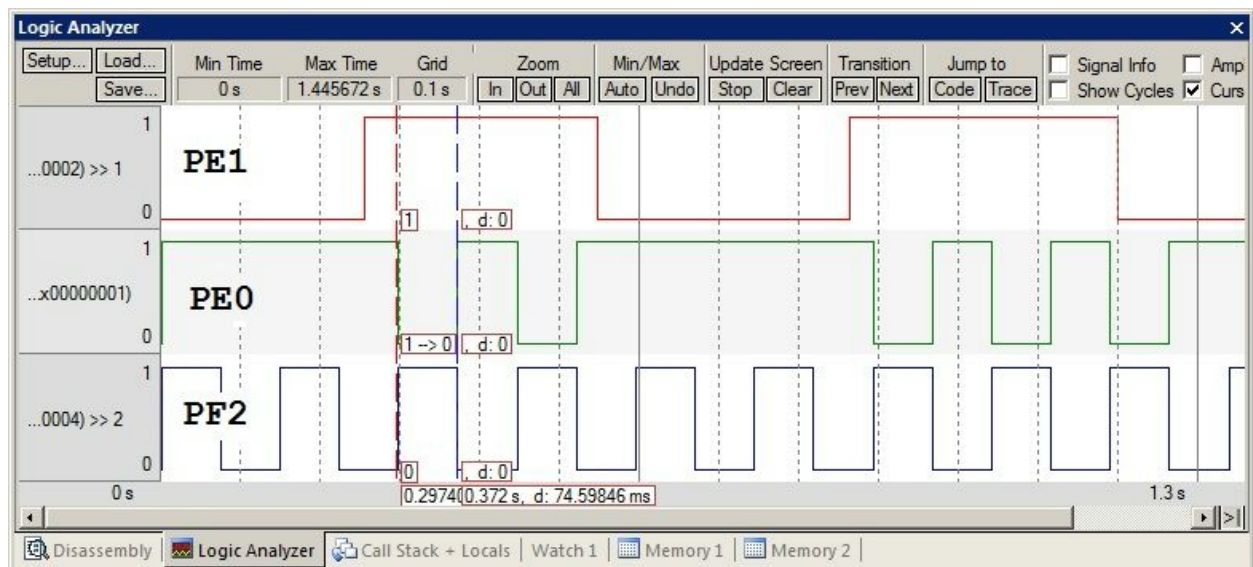
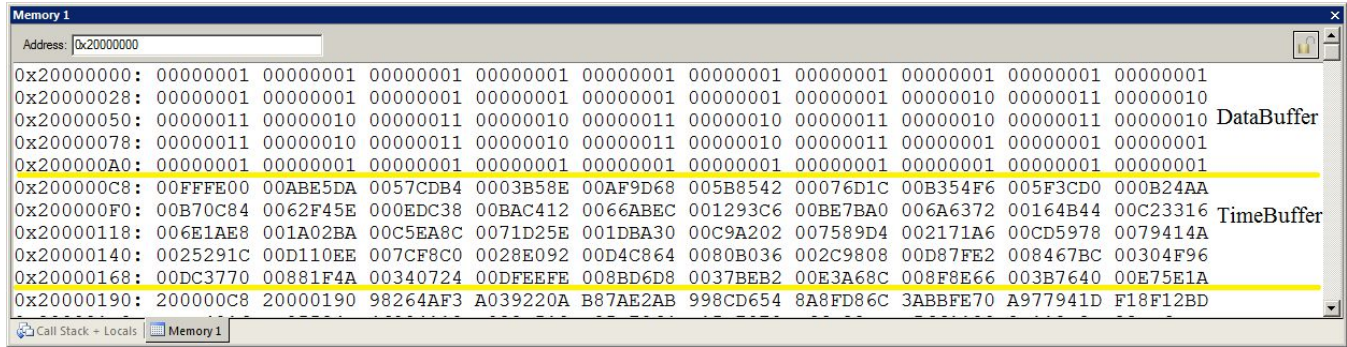


Figure 4.2. Simulation output showing the input on PE1, output on PE0 and heartbeat on PF2.



## Part e - 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 with a no-touch, touch, no-touch sequence during the 3-second measurement. Use the collected data to measure the period of the flashing LED. Your measurement should be accurate to 12.5 ns.

## 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. You should be able to connect PF2 to an oscilloscope to verify the main loop is running at about every 62 ms. 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:

- Why did you have to change the delay function after the PLL was activated?
- How did you change it?
- The TA will pick an instruction in your program and ask how much time does it take that instruction to execute in  $\mu\text{sec}$ . 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 of 8-bit data?
- If you used 8-bit data for **DataBuffer** what would be the advantages of 32-bit?
- 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?

Figure 4.3. Similar data is observed in the memory window in simulation and on the real board showing results of the dump.

- 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). Which of these values would constitute a software bug if it were to occur. How could you change **Debug\_Capture** to count the number of times this error state occurs.

## Deliverables

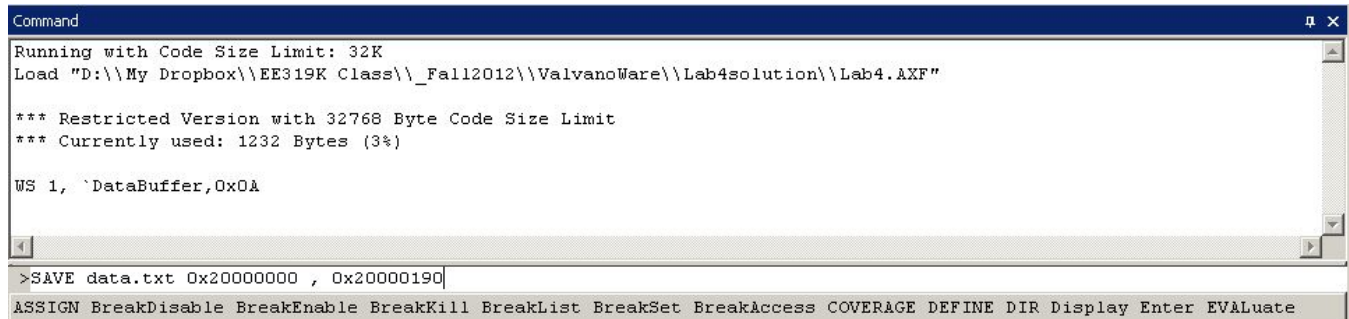
(Items 2, 3, 4, and 5 are one pdf file committed to SVN, have this file open during demo.)

- Lab 4 grading sheet. You can print it yourself or pick up a copy in lab. You fill out the information at the top.
- A screenshot showing the system running in simulation mode. In the screenshot, please show the dumped data in a memory window and the I/O window, as illustrated in Figures 4.2 and 4.3
- Assembly listing of your final program with both the dump and heartbeat instruments
- Estimation of the execution time of your debugging instrument **Debug\_Capture** (part b)
- Results of the debugging instrument (part e) and the calculation of the flashing LED period in msec.
- Optional Feedback : <http://goo.gl/forms/rBsP9NTxSy>

## Debugger output → file

How to transfer data from debugger to a computer file

- Run your system so data is collected in memory, assume interesting stuff is from 0x20000000 to 0x20000190
- Type **SAVE data.txt 0x20000000 , 0x20000190** in the command window after the prompt (“>”), type enter



```

Command
Running with Code Size Limit: 32K
Load "D:\My Dropbox\EE319K Class\_Fall2012\ValvanoWare\Lab4solution\Lab4.AXF"

*** Restricted Version with 32768 Byte Code Size Limit
*** Currently used: 1232 Bytes (3%)

WS 1, `DataBuffer,0x0A

>SAVE data.txt 0x20000000 , 0x20000190
ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet BreakAccess COVERAGE DEFINE DIR Display Enter EVALuate
  
```

- Open the **data.txt** file in NotePad, it looks like this

```

:0200000042000DA
:1000000001000000010000000100000001000000EC
:1000100001000000010000000100000001000000DC
:1000200001000000010000000100000001000000CC
:1000300001000000010000000100000001000000BC
:1000400001000000010000000110000000100000007E
:1000500011000000010000000110000000100000005E
:1000600011000000010000000110000000100000004E
:1000700011000000010000000110000000100000003E
:1000800011000000010000000110000000100000002E
:100090001100000001000000010000000100000004C
:1000A0000100000001000000010000000100000004C
  
```



```

:1000B000010000000100000001000000010000003C
:1000C000010000000100000000FEFF00DAE5AB00C7
:1000D000B4CD57008EB50300689DAF0042855B002C
:1000E0001C6D0700F654B300D03C5F00AA240B003F
:1000F000840CB7005EF4620038DC0E0012C4BA0053
:10010000ECAB6600C6931200A07BBE0072636A006F
:10011000444B16001633C200E81A6E00BA021A00E9
:100120008CEAC5005ED2710030BA1D0002A2C9007F
:10013000D4897500A67121007859CD004A41790013
:100140001C292500EE10D100C0F87C0092E02800A8
:1001500064C8D40036B0800008982C00E27FD80034
:10016000BC678400964F30007037DC004A1F88005F
:1001700024073400FEEEDF00D8D68B00B2BE370075
:100180008CA6E300668E8F0040763B001A5EE70087
:01019000C8A6
:00000001FF

```

- Strip off the first 9 characters of every line, and the last two characters of every line. Delete 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.

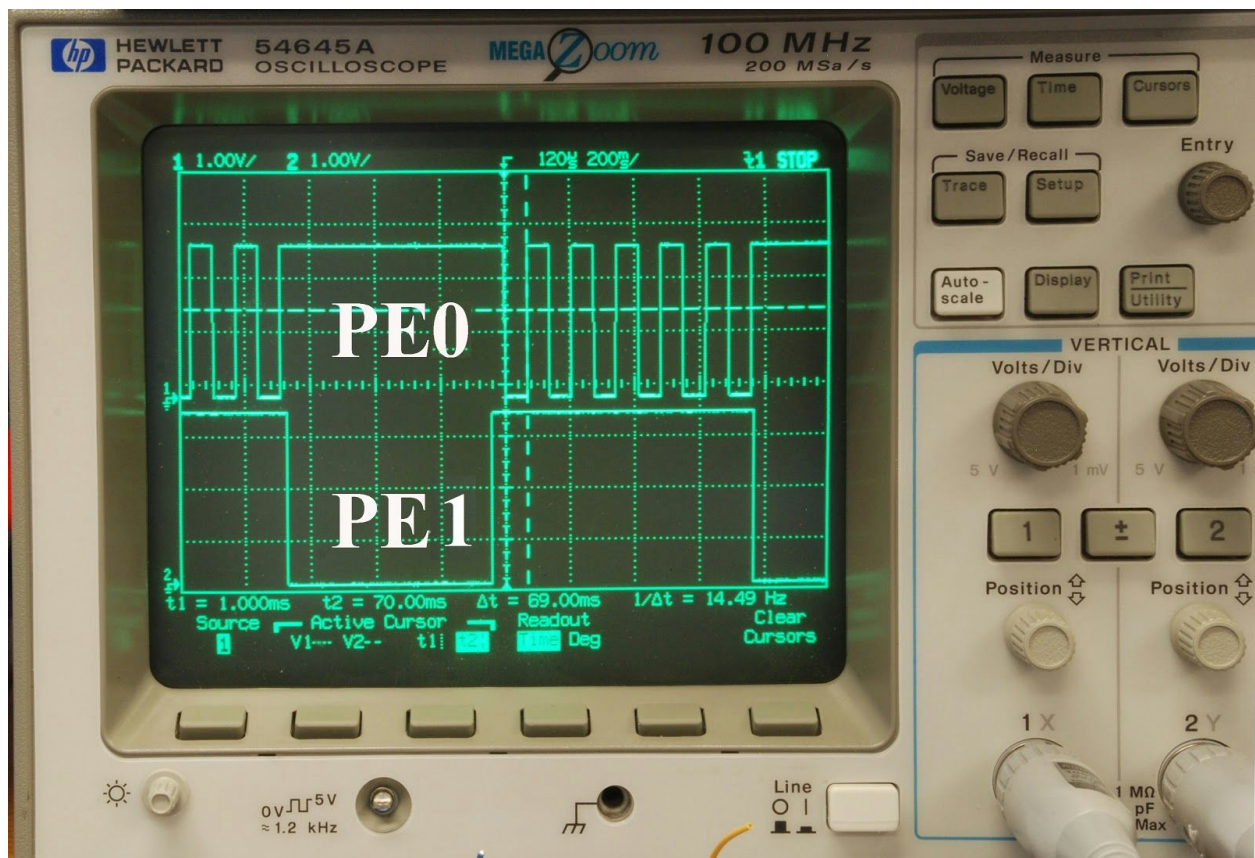




Figure 4.4. The input and output signals are measured with a two channel oscilloscope. When the switch is pressed, PE1 goes high. When the software sees PE1 high it toggles PE0 every 62ms. In this solution with system actually toggled PE0 every 69 ms.

## 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 array be located somewhere specific? How large is it supposed to be, and how can we make sure nothing else writes into that address?**

The "space" operator seen in the following code segment from the Lab4 starter file will allocate the amount of bytes to the right of the operator, and assign the label to the left as the address to the first byte. The AREA DATA, ALIGN=2 is placing the variables into RAM. The size depends on how much space you will need for your application. In this case, as the lab manual specifies, "... , if the outer loop of your Lab 3 executes in about 62 ms, then the loop will be executed about 50 times in 3 seconds (3000/62), so the array sizes will need to be 50 elements each." Keep in mind that in assembly, you will need to keep track of data types yourself. Since the TM4C is byte addressable, and we are storing full words, an element will be four bytes, so you will need to store 200 bytes total (hence the SIZE\*4). Nothing else will be placed in that data region by the assembler, but you will have to do bounds checking yourself to avoid overwriting information at run time.

```
                AREA    DATA, ALIGN=2
SIZE            EQU     50
;You MUST use these two buffers and two variables
;You MUST not change their names
DataBuffer     SPACE   SIZE*4
TimeBuffer     SPACE   SIZE*4
DataPt         SPACE   4
TimePt         SPACE   4
;These names MUST be exported
                EXPORT  DataBuffer
                EXPORT  TimeBuffer
                EXPORT  DataPt
                EXPORT  TimePt
```

**2. We're given the label for the Port E Pull-Up Resistor Register, but don't we need the one for Port F?**

You would use the PortF PUR register for the on board switch. However as an extension of Lab3, you will still be using PortE. Checkout the tm4c123gh6pm.h header file inside of the inc/ directory for all of the port addresses if you need one that's not in the starter file. For example GPIO\_PORTF\_PUR\_R is at 0x40025510

**3. 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."

**4. 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. 62ms 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.

**5. 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

**6. The SVN Lab 4 is set up slightly different than the original Lab 4 from the EE319K package files we downloaded. Which one are we supposed to use? The SVN one doesn't have NVIC\_ST\_CTRL\_R and NVIC\_ST\_RELOAD\_R declared. It also has ALIGN = 2. The other one has ALIGN = 4.**

The SVN version of main.s has all of the constants you need. The area statement might need to be modified to accommodate the double-word stack alignment requirement, but other than that should be fine.

**7. For the SysTick\_Init, do we have to import the code like by using IMPORT SysTick\_Init, or do we actually copy the code into our subroutine?**

I would use import to get access to the functions in SysTick.s. They should already be exported in the assembly file, so all you need to do is add "import SysTick\_Init" as you said, and you should be able to branch to the subroutine from your main.s (if you added SysTick.s to your project before hand). Right click on the "source" folder in your project navigator window on the left side, and select "add existing file". You can point to the SysTick.s file in the sample directory, but if you plan on modifying the file at all in the future, I would first copy over the assembly file into your project's directory first and select the copy. This will also keep all of your code in the same location, which is nice if you ever move your folders around or want to reference it again in the future.

**8. Do we need to use the SWITCH and LED label?**

The SWITCH and LED label are just there for code readability. They are bitbanded such that you can read and write to those addresses and only affect PE0 and PE1 accordingly. If you just used the GPIO\_PORTE\_DATA\_R, that's fine as well.

**9. 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.

**10. 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

**11. 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