EECS 3100 Embedded Systems

Lab 6 - Minimally Intrusive Debugging Methods

Original by: J. Valvano, et. al.
Adapted by: J. Debnath, T. Royko, and G. Serpen
Date: September 2016

Preparation (*to be completed prior to the lab session*)

- This is a team project.
- Read this project assignment document in its entirety.
- This lab will use the hardware prototype constructed for Lab 5.
- Read the document **Lab06-PreLabReading.docx.**
- You will need to study and understand the code for instantiating and initializing PLL and SysTick functionality, which is provided in the **SysTick_4C123asm** folder under the **…\ProjectAssignments\Lab – 06**. Open, compile, run and study the project code functionality to develop a good understanding of it.
- Complete the work for all parts in the section "**Procedure**" in this document.
- Keil project template is in the folder **…\Project Templates\Lab6_asm**
- Print a copy of and fill in your information in **Lab06-DemoRecord** sheet and bring it to the lab session.

Purpose

The purpose of this lab is to learn minimally intrusive debugging skills. When visualizing software running in real-time on an actual microcontroller, 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 program executes. 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 board.

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 verify experimentally the correct operation of your Lab 5 hardware prototype system. Specifically, 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.
- 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. For the current counter time, you simply read the 24-bit **`NVIC_ST_CURRENT_R`** value.
- 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 5 as your system runs in real time.
- Facilitate the **dump** to 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 later.
- Toggle an LED once each time through the loop to create a **heartbeat**.

Procedure

The basic approach 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. The grader is available for you to use while testing Lab 6. For example, the grader will measure the toggle rate, it will check for buffer initialization, and it will make simple tests to see if your system fills the buffer properly. However, in EECS 3100 your lab demonstration grade is determined by the TA during checkout.

Note the following:

- This lab adds onto the code you have written for Lab 5.
- All new code written for this lab must be in assembly.

Part A – Write Code for Dump Instrument

When your main program calls **`TExaS_Init`**, this subroutine will activate the **PLL** making bus clock frequency equal to 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 ise an array capable of storing about 3 seconds worth of **Port E** measurements, and a second array capable of storing about 3 seconds worth of time measurements. For example, if the outer loop of your Lab 5 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 counters or pointers 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 DataBuffer and TimeBuffer 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`** value 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 position and the PE0 value into bit 0 position. Table 6.1 below illustrates how this makes the data easier to visualize after a dump is taken.

Input (PE1)	Output (PE0)	Saved 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

Table 6.1. Example input output dump table

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 memory location/address to save the I/O data and its timestamp.

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 1 into bit 4 position, and leave bit 0 in 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 registers saved and return

For regular functions, we are free to use registers 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 and also when not pressed. The dumped data should start with some 0x01 values, 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 Instrument

A heartbeat is a quick and convenient way to see if your program is still running. Write debugging software and add **PF2** as an output to the Lab 5 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 all labs 6 and beyond.

Although not for this project, you may, for future projects, need to 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 EECS 3100 there will be one heartbeat on **PF2**, and the heartbeat must occur slow enough to be seen with the naked eye.

Part D – Perform Debugging

Debug your combined hardware/software system first with the simulator, then on the actual TM4C123 board. Figure 6.1 shows how to configure the Lab 6 simulator, and Figure 6.2 is the TExaS emulator GUI. Figure 6.3 is screenshot of Lab 6 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 6.4 shows in simulation the contents of the two buffers containing the information for debugging purposes.

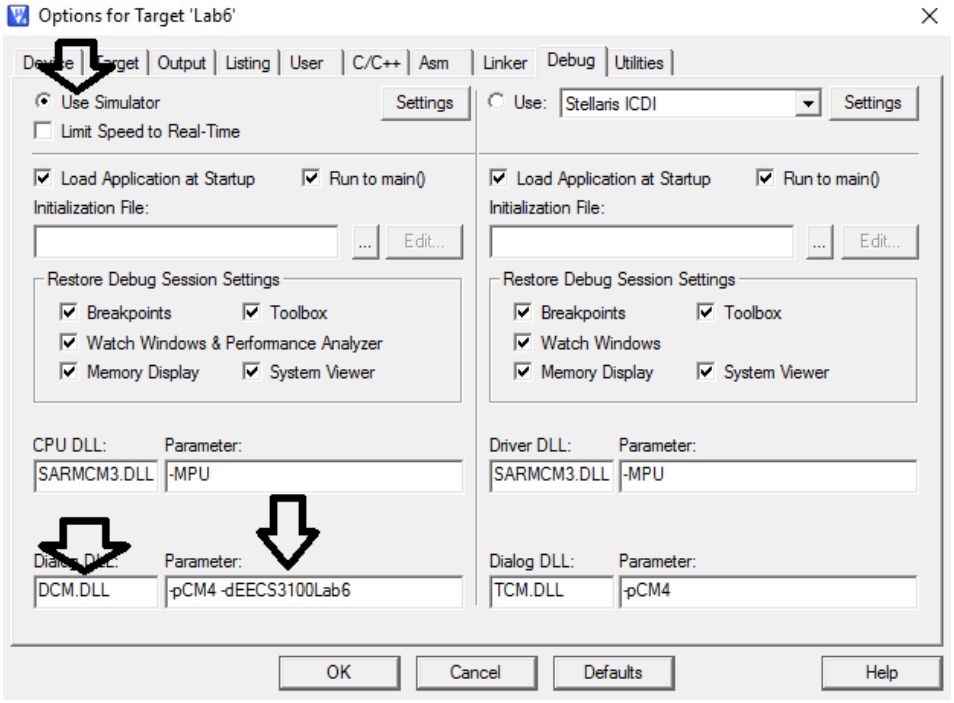


Figure 6.1. Options Dialog.
(DCM.DLL -pCM4 -dEECS3100Lab6)

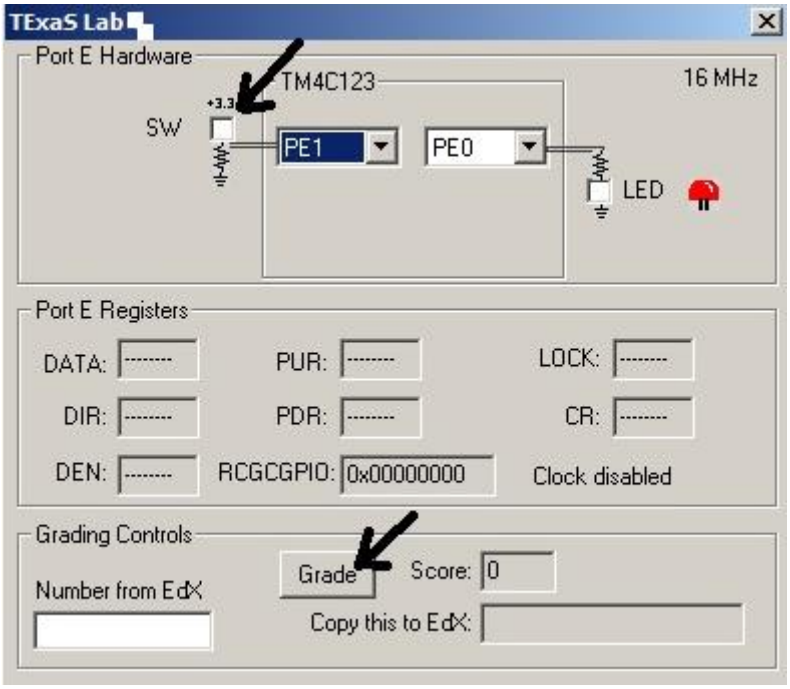


Figure 6.2. Using TExaS to debug your software in simulation mode

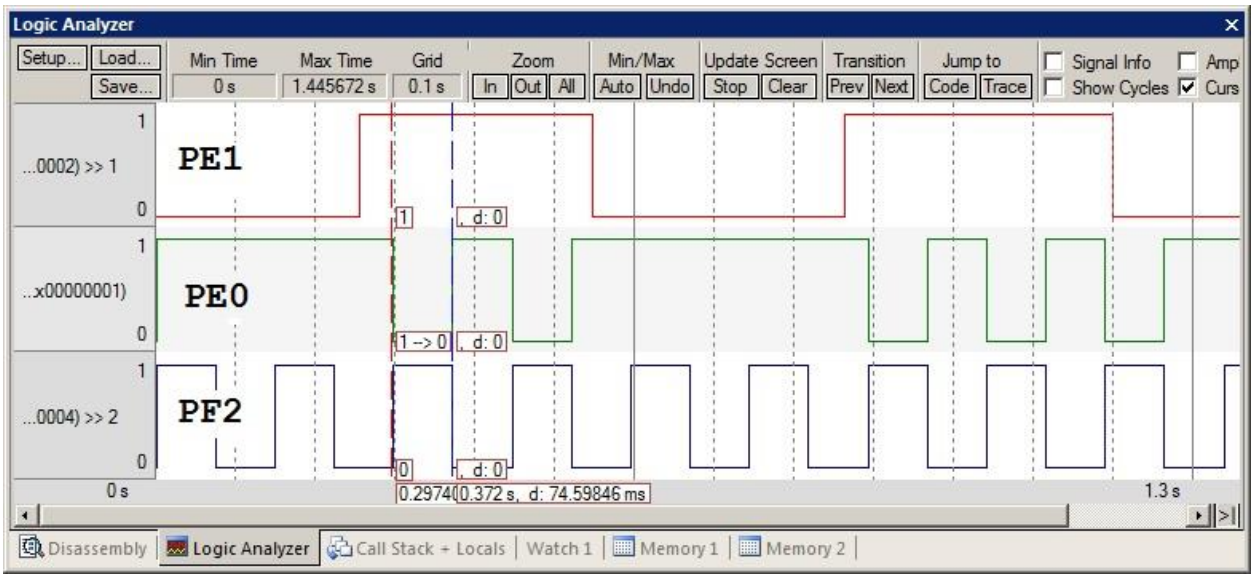


Figure 6.3. Simulation output showing the input on PE1, output on PE0 and heartbeat on PF2.

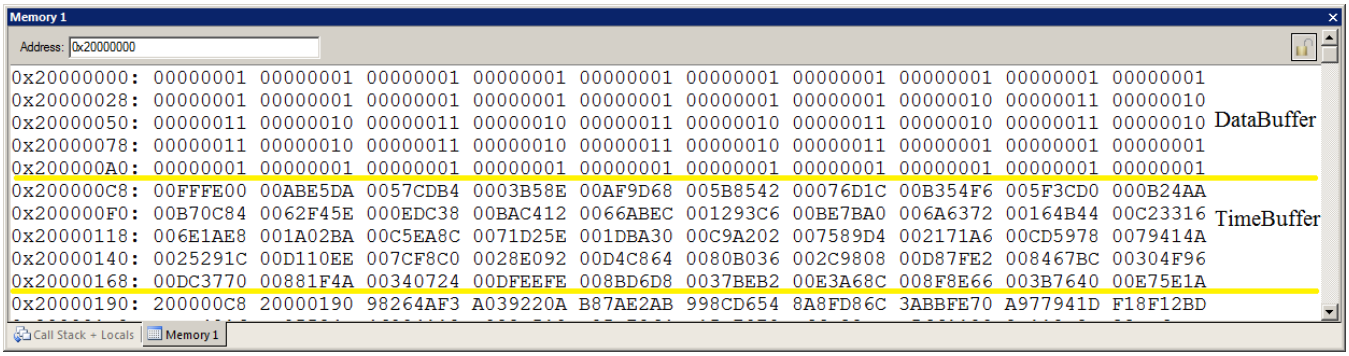


Figure 6.4. Similar data is observed in the memory window in simulation and on the real board showing results of the dump.

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 (Figure 6.5) 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:

1. Why did you have to change the delay function after the PLL was activated?
2. How did you change it?
3. 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?
4. You will be asked to create a breakpoint, and add the port pin to the simulated logic analyzer.
5. What do you mean by intrusiveness?
6. Is **`Debug_Capture`** minimally intrusive or non-intrusive?
7. Is your code "friendly"?
8. How do you define masking?
9. How do you set/clear one bit in without affecting other bits?

10. What is the difference between the **B**, **BL** and **BX** instructions?
11. How do you initialize the SysTick?
12. You should understand every step of the function **SysTick_Init**.
13. How do you change the rate at which SysTick counts?
14. Describe three ways to measure the time a software function takes to execute?
15. How do you calculate the sizes of the port data and the timestamp data?
16. If you used 32-bit data for **DataBuffer** what would be the advantages of 8-bit data?
17. If you used 8-bit data for **DataBuffer** what would be the advantages of 32-bit?
18. Could you have stored the time-stamp data in 8-bit, 16-bit, or 24-bit arrays?
19. 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?
20. How do you allocate global variables?
21. 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?

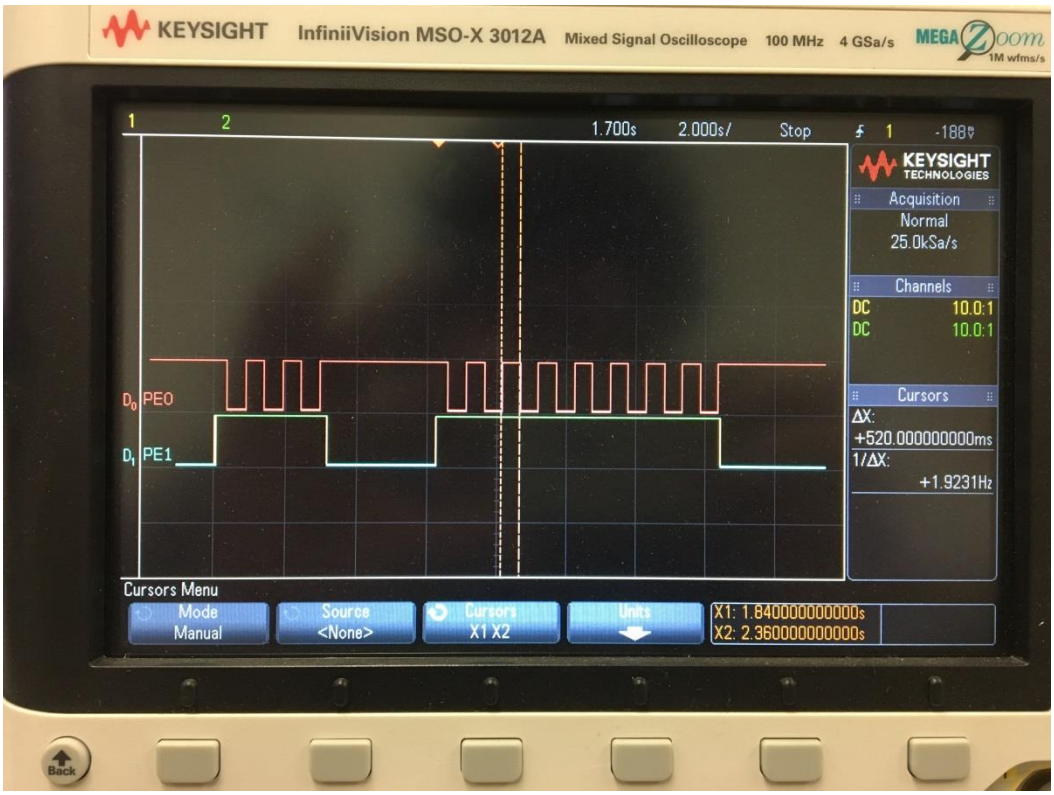


Figure 6.5. The input and output signals are measured with a two channel oscilloscope. When the switch is pressed, PE1 goes high. When the software sees PE1 high it toggles PE0 every 62ms. In this solution, system actually toggled PE0 every 69 ms.

Deliverables

Please submit a written project report: refer to the **Lab06-GradingChart** document to prepare your lab report and include the following as attachments:

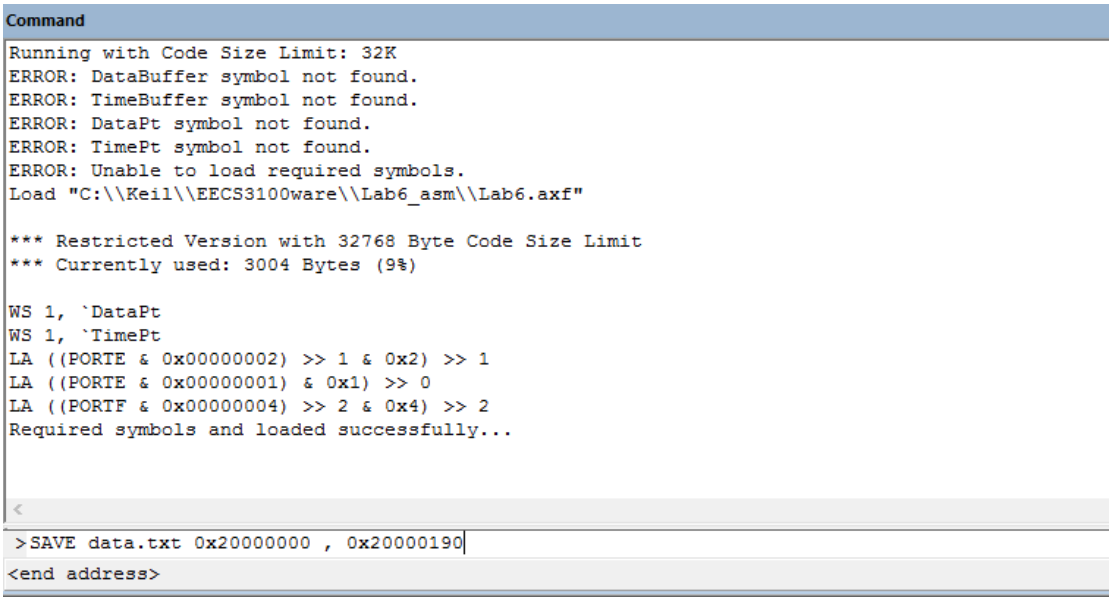
1. A screenshot showing the system running in simulation mode. In the screenshot, show the dumped data in a memory window and the I/O window, as illustrated in Figures 6.3 and 6.4.
2. Assembly listing of your final program with both the dump and heartbeat instruments.
3. Estimation of the execution time for your debugging instrument **`Debug_Capture`** (see Part B in the Procedure section).
4. Results of the debugging instrument and the calculation of the flashing LED period in msec (Part E in the Procedure section).
5. Printout of data.txt.
6. A narrative for detailed description of team member contributions to all aspects of the project including hardware prototyping design and test, code development, software test and debugging, and report preparation and authoring.

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 0x20000000 to 0x20000190.

2. Type **SAVE data.txt 0x20000000, 0x20000190** in the command window after the prompt (“>”), type enter



3. Open the **data.txt** file in NotePad: it looks like this:

```
:020000042000DA
:10000000010000000100000001000000EC
:10001000010000000100000001000000DC
:10002000010000000100000001000000CC
:10003000010000000100000001000000BC
:10004000010000001100000010000000 7E
:10005000110000001100000010000000 5E
:10006000110000001100000010000000 4E
:10007000110000001100000010000000 3E
:10008000110000001100000010000000 2E
:10009000110000001000000010000000 4C
:1000A000010000000100000001000000 4C
:1000B000010000000100000001000000 3C
:1000C000010000000100000000FEFF00DAE5AB00C7
:1000D000B4CD57008EB50300689DAF0042855B002C
:1000E0001C6D0700F654B300D03C5F00AA240B003F
:1000F000840CB7005EF4620038DC0E0012C4BA0053
```

```
:10010000ECAB6600C6931200A07BBE0072636A006F
:10011000444B16001633C200E81A6E00BA021A00E9
:100120008CEAC5005ED2710030BA1D0002A2C9007F
:10013000D4897500A67121007859CD004A4179013
:100140001C292500EE10D100C0F87C0092E02800A8
:1001500064C8D40036B0800008982C00E27FD80034
:10016000BC678400964F30007037DC004A1F88005F
:1001700024073400FEEEDF00D8D68B00B2BE37075
:100180008CA6E300668E8F0040763B001A5EE70087
:01019000C8A6
:00000001FF
```

4. 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.

FAQ

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 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 project assignment document specifies, "..., if the outer loop of your Lab 5 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 Lab5, you will still be using PortE. Check out 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. *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 80 MHz. Further, we did ask you to implement a minimally intrusive debugging instrument. Therefore, 0.0011% qualifies as minimally intrusive.

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. *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.

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

   The SWITCH and LED label are just there for code readability. They are bit-specific addresses, 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 is fine as well.

8. *We have the values, 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.

9. *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

10. *After I debug my code this error appears:* Error: Could not load file C:\Keil\...\Lab6.axf. Debugger aborted!

    That error message is telling you that you do not have an executable file (*.axf) and it therefore cannot continue. Reasons that you do not have an executable file may include:
    - You did not compile/build
    - Last time you compiled/built the executable was not created due to errors