Lab 4. Minimally Intrusive Debugging (Sp 2021)

Due to the snowstorm of spring 2021, some parts of Lab4 are eliminated

Get new Lab4 starter project from https://utexas.instructure.com/courses/1304645/files/folder/unfiled

Unzip **Lab4.zip** into same place you are doing other labs (verify it builds and simulates)

- Preparation
- Purpose
- System Requirements
- Procedure
  - Part a - Write SysTick Initialization
  - Part b - Write Dump
  - Part c - Estimate Intrusiveness
  - Part d - Implement Heartbeat
  - Part e - Debug
  - Part f - Capture Timing
- Debugger output → file (this step is optional)
- Demonstration
- Deliverables

FAQ

Preparation

Read Book Sections 4.1, 4.2, 4.3, 4.4, 4.5, and 4.7
Open, compile, and run the projects

~~**SysTick_4C123asm** PLL and SysTick functions~~
Lab4_EE319K New starter project from EE319K canvas
https://utexas.instructure.com/files/58655006/download?download_frd=1
These videos https://www.youtube.com/playlist?list=PLyg2vmIzGxXHYEHQrxNxGcRg6vCTB20Ud relate to Lab 4. ~~This relates to SysTick https://youtu.be/ZYMYLNC5jmQ The code in the video is correct (the handwritten subtraction should have been R12=R3-R0)~~

Note: The direction of the motor changes if the switch is touched and released. In particular the direction cycles between these three modes: CW CCW and stopped. When spinning a stepper motor we must do two things. 1) The output must follow this pattern 5↔6↔10↔9↔5, forwards or backwards. 2) The time between outputs must be known and constant. The time between outputs sets the motor speed. The purpose of Lab 4 is to verify proper behavior of the stepper motor software.

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 functionality 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 a stepper motor control system that is given to you. The given system does work, but nevertheless, you are asked to debug it. You do not need to connect the stepper motor here in Lab 4, but you will need to interface a positive switch to PA2 input. The given system will use the switch to affect the direction of the motor rotation.

- Required Hardware connection: one external switch
    - PA2 is connected to positive logic momentary switch (internal pull down resistor)
- Hardware not actually built: stepper motor with 90 steps per rotation
    - PB3,PB2,PB1,PB0 are four digital outputs connected to a stepper motor

The first requirement of this lab is to verify the stepper motor output patterns are correct. The stepper motor is controled by four digital outputs, and the correct output pattern is ...,6,10,9,5,6,10,9,5… for spinning the motor in one direction and the reverse pattern (...,9,10,6,5,9,10,6,5…) for spinning the motor in the reverse direction. The output does not change to stop the motor (stays fixed at 5, 6, 10 or 9). More specifically, given the current 4-bit output following table defines the value 4-bit outputs that could be next.

Current Output	Next output to stop	Next output to step one direction	Next output to step other direction
5	5	6	9
6	6	10	5
10	10	9	6
9	9	5	10

~~The second requirement is that you will measure the time between the outputs in us and verify the time difference is approximately constant 11,111us (meaning the motor is spinning at a constant rate of 1 rps).~~

- ~~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. You 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. To measure time difference subtract the current time from the previous time. You will have to implement a 24-bit subtraction using the 32-bit arithmetic. For example, if the 24-bit counter is 0x000900 (2304), counts 0x1000 times (4096 times), the 24-bit counter becomes 0xFFF900. In 24-bit math 0x000900-0xFFF900 is correctly 0x1000. However, in 32-bit math 0x000900-0xFFF900 is incorrectly 0xFF00,1000. AND the difference with 0x00FFFFFF to implement 24-bit math. To convert to time in usec, divide the difference by 80.~~
- Instrumentation data to be gathered is as follows: Every time a new output is written to the stepper motor record one 8-bit entry in the Data Buffer and one 32-bit entry in the Time Buffer:
    - The Data Buffer entry (byte) content has:
        - Bits 3,2,1,0 are the new output written to PB3,PB2,PB1,PB0
        - Bit 4 contains the current state of Button (PA2), in positive logic
    - ~~The Time Buffer entry (32-bit) has:~~
        - ~~Time difference between the current time and the time of the last output~~
        - ~~The times will have units of us~~
        - ~~The first recording will be incorrect (we do not care)~~
    - The size of both buffers is 100 entries.
        - Once you fill these entries you should stop collecting data

The third requirement is to add a debugging heartbeat to PF3, green LED on LaunchPad. The purpose of the heartbeat is to verify the software is running.

- The Heartbeat is an indicator of the running software. The Heartbeat debugging instrument (toggle the LED) is called once each run through the main loop to indicate that the system is live (not stuck or dead).

Procedure

The basic approach to Lab 4 through Lab 8 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 for this task. You have to 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 **NVIC_ST_CURRENT_R** as an indication of the current time.~~

Part b - Write Debugging Dump

When your main program calls **`TExaS_Init`**, this subroutine will activate the **PLL** making the bus clock 80 MHz. ~~So all **NVIC_ST_CURRENT_R** measurements use a 12.5ns clock cycle.~~ Write two debugging subroutines, **Debug_Init** and **Debug_Capture**. These routines are called debugging instruments. 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 100 entries of I/O data (5-bit), and a second array capable of storing 100 entries of time measurements (32-bit).

The first subroutine (**Debug_Init**) initializes your dump instrument. The initialization should activate the SysTick timer, place 0xFF into all locations of the data array and 0xFFFFFFFF into all locations of the time array to signify that no data has been saved yet. You will also initialize pointers and/or counters as needed. The second subroutine (**Debug_Capture**) that saves one data-point (**PA2** input data, and **PB3-0** output data) in the data array and one time difference measurement (elapsed time in usec from previous call to capture and this call to capture) in the time array.

We have already placed the call to **Debug_Init** at the beginning of the system. We have placed the call to **Debug_Capture** each time the existing stepper motor controller writes to the stepper motor. 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 100 entries of state data)
2. ~~Allocate **TimeBuffer** in RAM (to store 100 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, assuming a pointer scheme

1. Save all registers used (because this is a debugging instrument, not a user function)
2. Return immediately if the buffers are full
3. Read Ports A and B ~~and the SysTick timer (**NVIC_ST_CURRENT_R**)~~
4. Mask/shift, combining just bits PA2,PB3,PB2,PB1,PB0 into one 8-bit data value
5. Dump this data information into **DataBuffer** using the pointer **DataPt**
6. Increment **DataPt** to next address
7. ~~Calculate the elapsed time in usec (difference between now and previous Capture)~~
8. ~~Dump elapsed time into **TimeBuffer** using the pointer **TimePt**~~
9. ~~Increment **TimePt** to next address~~
10. Restore registers as needed 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 that are modified, 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 permanently in RAM. You can observe the debugging arrays using `Memory` windows. 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. Because the **Debug_Capture** subroutine once per main loop, the time between calls will be about 50ms. Calculate the percentage overhead required to run the debugging instrument (time to execute instrument divided by time between calls to the instrument). 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 function to toggle PF3 (green onboard LED). We have placed the call to **`Debug_Beat`** at the beginning of the main loop, indicating the software is running. Since the function is called 90 times a second, you will need to slow it down by toggling the pin every 11 calls. In particular, the function is called every 11.1 ms, but the LED should toggle every 122ms. Do not add a wait, which would be highly intrusive. Rather add a counter to determine if it has been 11 calls since the last toggle. A heartbeat of this type will be added to all software written for Labs 5, 6, 7, 8, and 9 in this class. A heartbeat is a quick and convenient way to see if your program is still running.

In more complicated systems (e.g., Ethernet router) you can implement multiple heartbeats at various strategic places in multiple places in the software. Sometimes the toggle rate is 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, 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 or on the actual TM4C123 board. If you run on the real board, you will need to add a positive logic switch to the PA2 input with the internal pull down resistor. You can find the address of the buffers by looking in the map file. Take screen shots to verify your system is working properly

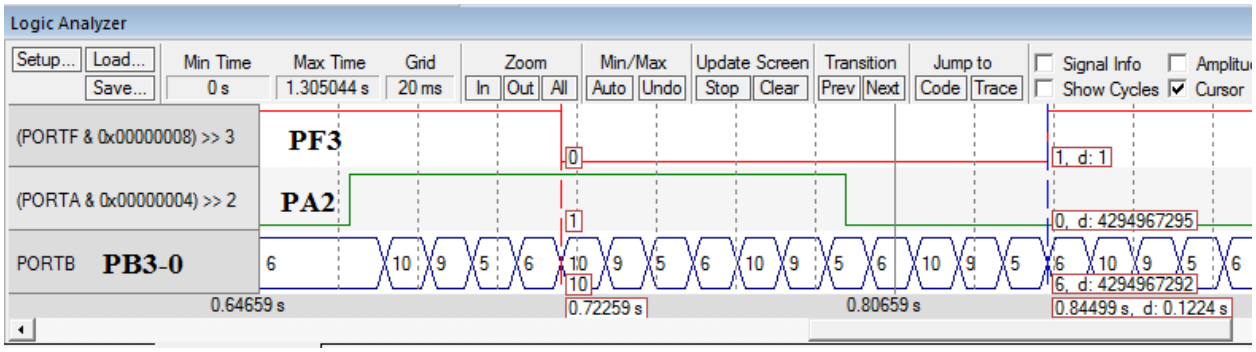


Figure 4.2. Simulation output showing heartbeat on PF3, the input on PA2, and output on PB3-0.

The figure (4.3) below shows the memory window for both the DataBuffer and TimeBuffer. Notice the elapsed time are about 11.111ms, creating a motor speed of 1 rps (this motor has 90 steps per rotation).

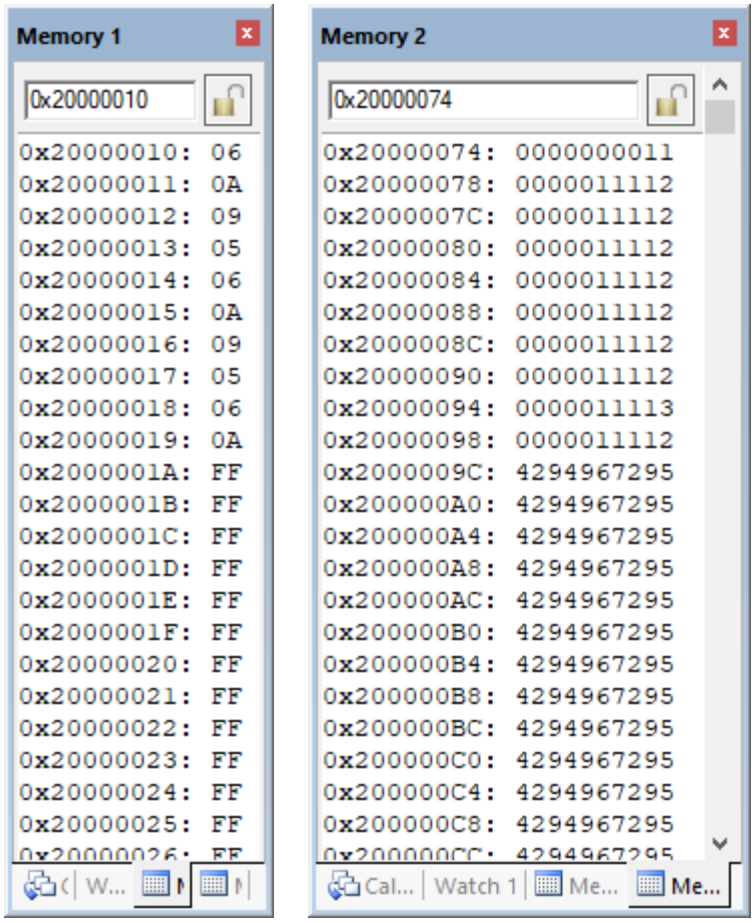


Figure 4.3. State ~~and Timings~~ are observed in the respective memory windows on the real board showing results of the dump after the first 10 entries were made. ~~The time measurements will be different on the simulator.~~ Check you map file to see address for your DataBuffer ~~and TimeBuffer~~

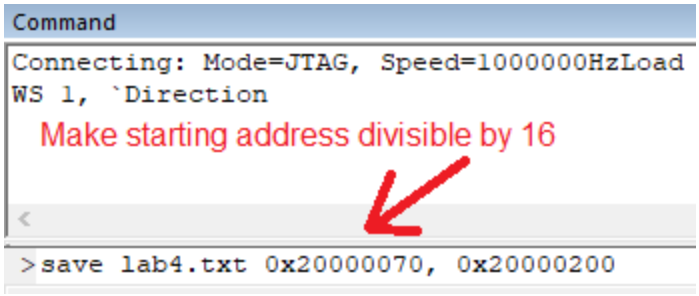
Part f - Capture Timing

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

Debugger output → file (this step is optional)

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 0x20000074 to 0x20000200; Make starting address divisible by 16, Note the last entry ends at 0x20000203



2. Type **SAVE lab4.txt 0x20000070 , 0x20000200** in the command window after the prompt (“>”), type enter. The file is saved in the project folder (same folder as Lab 4 files)
3. Open the **lab4.txt** file in NotePad, it is in a standard format called the Intel Hex format(we encourage you to read the format documentation). Here is how it looks:

```
:020000042000DA
:10007000060A0905**0B000000682B0000682B0000**31
:10008000**682B0000682B0000682B0000682B0000**24
:10009000**682B0000692B0000682B0000682B0000**13
:1000A000**682B0000682B0000682B0000682B0000**04
:1000B000**682B0000692B0000682B0000682B0000**F3
:1000C000**682B0000682B0000682B0000682B0000**E4
:1000D000**682B0000692B0000682B0000682B0000**D3
:1000E000**682B0000682B0000682B0000682B0000**C4
:1000F000**682B0000692B0000682B0000682B0000**B3
:10010000**682B0000682B0000682B0000682B0000**A3
:10011000**682B0000692B0000682B0000682B0000**92
:10012000**682B0000682B0000682B0000682B0000**83
:10013000**682B0000692B0000682B0000682B0000**72
:10014000**682B0000682B0000682B0000682B0000**63
:10015000**682B0000692B0000682B0000682B0000**52
:10016000**682B0000682B0000682B0000682B0000**43
:10017000**682B0000692B0000682B0000682B0000**32
:10018000**682B0000682B0000682B0000682B0000**23
:10019000**682B0000692B0000682B0000682B0000**12
:1001A000**682B0000682B0000682B0000682B0000**03
:1001B000**682B0000692B0000682B0000682B0000**F2
:1001C000**682B0000682B0000682B0000682B0000**E3
:1001D000**682B0000692B0000682B0000682B0000**D2
:1001E000**682B0000682B0000682B0000682B0000**C3
:1001F000**682B0000692B0000682B0000682B0000**B2
:010200006895
:00000001FF
```

4. 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 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 **lab4.txt** file for automatic analysis into the excel file (**Calculation_Sp21.xlsx**). The excel file should be self-explanatory.

Demonstration

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

There are grading sheets for every lab so you know exactly how you will be evaluated. You will show the TA your program operation on **either the simulator or** 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 the 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?
- Why did we make you toggle the green LED every 11th call to Capture_Beat?

Do all these well in advance of your checkout

1. **Signup for a Zoom time with a TA. If you have a partner, then both must be present**
2. **If you cannot live-stream video, create a 60-sec YouTube video and upload it**
3. **Upload your software to canvas, make sure your names are on all your software**
4. **Upload your one pdf with deliverables to Canvas**

Do all these during Zoom meeting

1. **Have your one pdf with deliverables open on your computer so it can be shared**
2. **Have Keil Lab 4 open so TA can ask about your code**
3. **Start promptly, because we are on a schedule. If you have a partner, then both must be connected**
4. **Demonstrate lab to TA (YouTube video or livestream video)**
5. **Answer questions from TA in the usual manner**
6. **TA tells you your score (later the TA will upload scores to Canvas)**

Deliverables

Upload your main.s file to Canvas. Combine the following components into one pdf file and upload this file also to Canvas. UPLOAD ONLY ONE COPY PER TEAM (names on both). Have the pdf file and Keil open on the computer during demonstration

1. Your names, professors, and EIDs.
2. 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
3. Estimation of the execution time of your debugging instrument **`Debug_Capture`** (part b)
4. Results of the debugging instrument (part e) ~~and the calculation of the minimum and maximum elapsed time (ignore the first measurement, which will be wrong).~~

Optional Feedback : http://goo.gl/forms/rBsP9NTxSy

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 100
   TimeBuffer SPACE 4*100
   DataPt SPACE 4
   TimePt SPACE 4
   ```

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

2. **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."

3. **So I'm getting a percentage overhead of 0.011%. Does that seem reasonable?**
   The overhead will be dominated by how much time you delay for in your loop. 11.1ms is a *ton* of time when the clock ticks at 80MHz. Further, we did ask you to implement a minimally intrusive debugging instrument. 0.011% sounds minimally intrusive to me.

4. **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

5. **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.

6. **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 the output to the stepper must follow the 5,6,10,9 sequence. If the current output is 5, then the next output could be 5(no motion), 9 (one step CCW), or 6 (one step CW). If the current output is 6, then the next output could be 6(no motion), 5 (one step CCW), or 10 (one step

CW). If the current output is 10, then the next output could be 10 (no motion), 6 (one step CCW), or 9 (one step CW). If the current output is 9, then the next output could be 9(no motion), 10 (one step CCW), or 5 (one step CW). Each step is a fixed angle, so the time between steps determines rotational speed.

7. **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

8. **After I debug my code this error appears: Error: Could not load file 'C:\Keil\EE319Kware\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

9. **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. This type of debugging can be inserted into the delivered system so the system can make self-checks after it is deployed into the field (without having the debugger attached, or a logic analyzer used)

10. **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 outer while loop.

11. **What is the excel file for?**
    Remember, the excel file is optional. 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 lab4.txt that keil produces using the SAVE command (does the unpacking of Intel hex format).

12. **What is the TExaSdisplay logic analyzer?** With the USB cable connected, run your system with the TExaS display window open and you can see the 6 pins in real time. Ask your TA to show you

how it works. After calling TExaS_Init data is sent from LaunchPad to PC 10,000 times/sec.