# Lab 2 Performance Debugging

This laboratory assignment accompanies the book, Embedded Systems: Real-Time Interfacing to ARM Cortex M Microcontrollers, ISBN-13: 978-1463590154, by Jonathan W. Valvano, copyright © 2015.

**Goals**
- To develop software debugging techniques,
  - Performance debugging (dynamic or real time)
  - Profiling (detection and visualization of program activity)
- To dump time and data values into arrays
- To learn how to use the oscilloscope and logic analyzer,
- To experience concepts of real time, probability mass function and Central Limit Theorem
- To observe critical sections,
- Get an early start on Lab 3, by writing a line drawing function.

**Review**
- Valvano Section 2.4 on GPIO, Chapter 10 of data sheet
- Valvano Sections 3.9, 5.9 on debugging,
- Valvano Section 5.3 on critical sections,
- Valvano Section 6.2 on periodic timer interrupts, Chapter 11 of data sheet
- Valvano Section 8.5 on the ADC, Chapter 13 of data sheet
- Logic analyzer instructions.

**Starter files**
- **ADCSWTrigger_4C123** **PeriodicTimer1AInts_4C123** and your Lab 1

**Background**

In this lab we will develop debugging techniques to experience fundamental concepts of real time, critical sections, probability mass function (PMF), and the Central Limit Theorem (CLT). You should review real-time, time jitter, and critical sections from the book. Do an internet search of PMF and CLT.

**Preparation (do this before lab starts)**

1. Make a copy of the **ADCSWTrigger_4C123** project and call it Lab 2. Compile and run the project either on the board or on the simulator. Observe the variable **`ADCvalue`** and the flashing of the LEDs on PF2 and PF1.

2. The microcontroller is executing at 80 MHz. Assuming assembly instructions average about 2 cycles per instruction, it takes about 25 ns to execute an instruction. Look at the following C code and associated assembly created by the compiler. Answer the following questions

a) What is the purpose of all the DCW statements?
b) The main program toggles PF1. Neglecting interrupts for this part, estimate how fast PF1 will toggle.
c) What is in R0 after the first LDR is executed? What is in R0 after the second LDR is executed?
d) How would you have written the compiler to remove an instruction?
e) 100-Hz ADC sampling occurs in the Timer0 ISR. The ISR toggles PF2 three times. Toggling three times in the ISR allows you to measure both the time to execute the ISR and the time between interrupts. See Figure 2.1. Do these two read-modify write sequences to Port F create a critical section? If yes, describe how to remove the critical section? If no, justify your answer?

*This assembly code was obtained by observing the assembly listing in the debugger. You may see different assembly on your machine because of differences in the compiler version or optimization settings. You are allowed to solve the preparation with either this assembly or the assembly you see on your computer.*

```
0x00000672 4806      LDR r0,[pc,#24]   ; @0x0000068C
0x00000674 6880      LDR r0,[r0,#0x08]
0x00000676 F0800002  EOR r0,r0,#0x02
0x0000067A 4904      LDR r1,[pc,#16]   ; @0x0000068C
0x0000067C 6088      STR r0,[r1,#0x08]
0x0000067E E7F8      B   0x0000067E
0x00000680 E608      DCW 0xE608
0x00000682 400F      DCW 0x400F
0x00000684 5400      DCW 0x5400
0x00000686 4002      DCW 0x4002
0x00000688 551C      DCW 0x551C
0x0000068A 4002      DCW 0x4002
0x0000068C 5000      DCW 0x5000
0x0000068E 4002      DCW 0x4002
```

```
while(1){
  PF1 ^= 0x02;
}
```

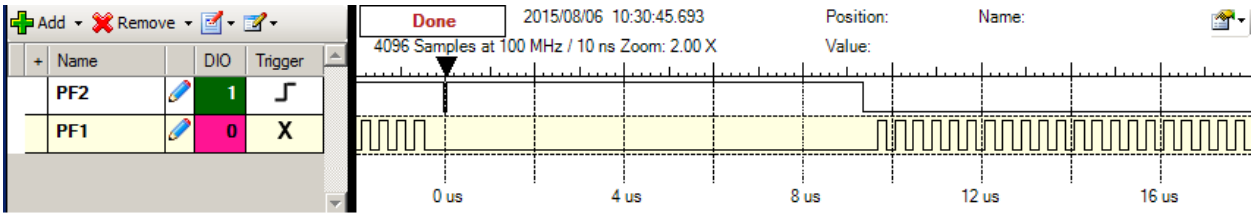


*Figure 2.1. Zoomed in view of the PF1 PF2 recording to see a) the main program does not run while the ISR is running and b) the time to execute the ISR is about 10us (most of this 10us occurs converting the ADC).*

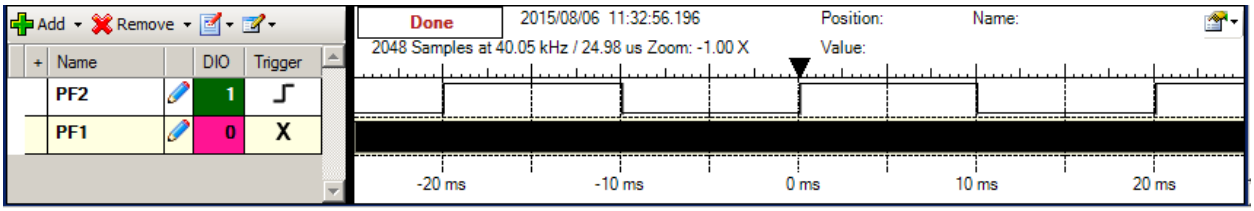


*Figure 2.2. Zoomed out view of the PF1 PF2 recording to see a) the ISR runs every 10ms and b) most of the processor time is allocated to running the main program.*

3. Write software that initializes one of the 32-bit timers to count every 12.5ns continuously. Basically copy the initialization from the **PeriodicTimer1AInts_4C123** project, set the **TIMER1_TAILR_R** to 0xFFFFFFFF, and remove the interrupt enable and interrupt arm statements. This way, reading the Timer 1 **TIMER1_TAR_R** will return the current time in 12.5ns units. The timer counts down. To measure elapsed time we read **TIMER1_TAR_R** twice and subtract the second measurement from the first. 12.5ns*$2^{32}$ is 53 seconds. So this approach will be valid for measuring elapsed times less than 53 seconds. The time measurement resolution is 12.5 ns.

4. Define two arrays of 1000 entries each. Add debugging dumps to the Timer0 ISR to record the time and ADC data value for the first 1000 ADC measurements. At 100 Hz sampling, these arrays will fill in 10 seconds. Once the arrays are full, stop recording. If you wish to run faster you could increase the sampling rate to 1000 Hz.

5. Write main program software to process the time recordings. This software will be run after the arrays are full (and not during the data collection phase.) Calculate 999 time differences and determine time jitter, which is the difference between the largest and smallest time difference. The objective is to start the ADC every 10 ms, so any time difference other than 10 ms is an error. Real-time ADC sampling requires time jitter to be small.

6. Write main program software to process the data recordings. This software will be run after the arrays are full (and not during the data collection phase.) We will assume the analog input is fixed, so any variations in data will be caused by noise. Create a probability mass function of the measured data. The shape of this curve describes the noise process. The x-axis has the ADC value and the y-axis is the number of occurrences of that ADC value.

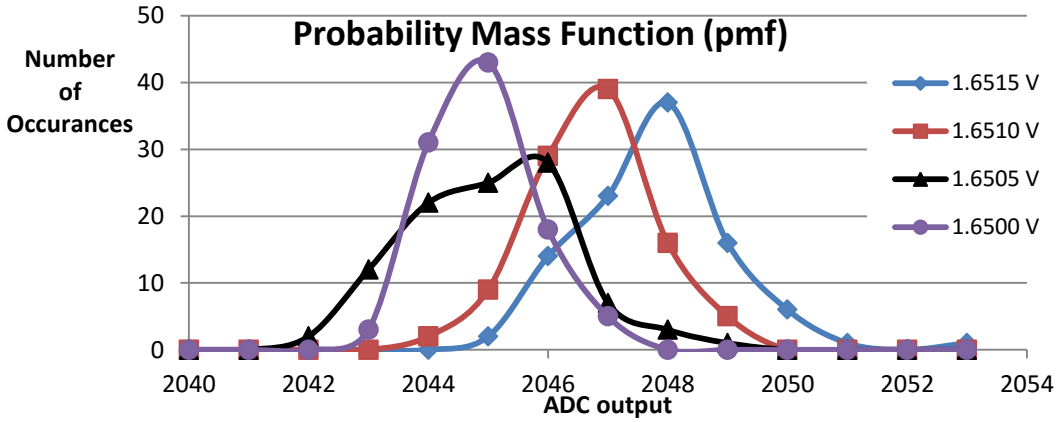


*Figure 2.3. Probability mass function measured on the TM4C123 ADC with 64-point averaging.*

**Procedure (do this during lab)**

**A. Learn how to use an oscilloscope**

You are expected to learn how to use an oscilloscope in this class, so, please ask your TA for a demonstration if you are unfamiliar with the features of the scopes we have in lab. In particular, you should: 1) be able to adjust the time base and voltage scales; 2) know how to set/adjust the trigger; 3) understand AC/DC mode; 4) be able to measure a frequency spectrum; 5) understand the resistive and capacitive load of the scope probe; 6) pulse width measurement using time cursors; 7) measure voltage amplitude using the voltage cursors; and 8) be able to save waveforms to USB flash drive for printout later. Line trigger mode is very useful for identifying the presence of 60 Hz AC-coupled noise.

*Deliverable*: Use a two channel scope to measure the debugging profile like Figure 2.1, and use the scope to estimate the time required to take one sample (interrupt, ADC, return from interrupt). Place a picture of the scope trace (photo or digital download) into your lab manual.

**B. Learn how to use a logic analyzer**

You are expected to learn how to use a logic analyzer in this class, so, please ask your TA for a demonstration if you are unfamiliar with the features of the logic analyzers we have in lab.

*Deliverable*: Use the logic analyzer to measure the debugging profile like Figures 2.1 and 2.2, and use the logic analyzer to estimate percentage of time running in the main versus running in the ISR. Place a picture of the scope trace (photo or digital download) into your lab manual.

**C. Experience a critical section**

In the main program change this line

```
PF1 ^= 0x02;  // toggles when running in main
```

to this line (creating a critical section)

```
GPIO_PORTF_DATA_R ^= 0x02;  // toggles when running in main
```

Use either the scope or the logic analyzer to observe the critical section. You should see something like Figure 2.4.

*Deliverable*: Which pin is incorrect PF1 or PF2? Look at the assembly language of the main program and explain the sequence of steps that results in the corruption of the debugging profile. Other than using bit specific addressing (PF1 PF2) what other solutions could you have used to have two debugging profile pins without a critical section?

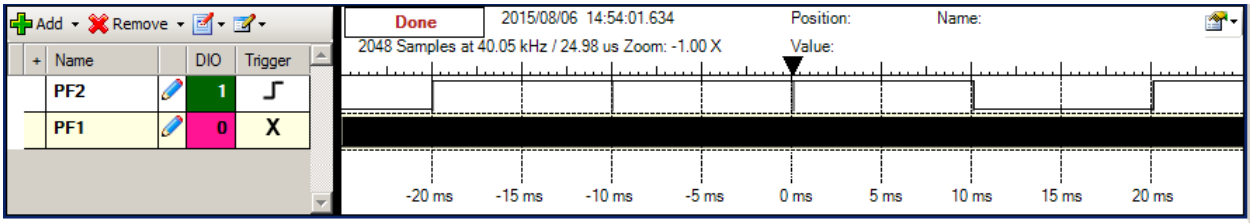


*Figure 2.4. Zoomed out view of the PF1 PF2 recording illustrating the results of the critical section (compare to Figure 2.2).*

**D. Prove the system is real time**

Debug your time dump and jitter calculation as written in preparations 4 and 5. Perform two or three runs to see if you always get the same time jitter measurement. With just one interrupt active, any jitter you do get (you might not see any) is caused by the random event of which instruction in the main program was being executed at the time of the interrupt trigger. If you want to see this single interrupt system have jitter place some code in the main program loop that execute an integer divide (the divide instruction takes 2 to 12 cycles):

```
PF1 = (PF1*12345678)/1234567+0x02;  // this line causes jitter
```

*Deliverable*: Measure time jitter with just the one sampling interrupt active.

This Timer 0 ISR is running at priority 2. Add one or two other 10kHzperiodic interrupts (SysTick Timer2 or Timer3) running at priority 1. Make the period close to 10 kHz but not exactly equal to 10 kHz (I ran mine at 99 us period). Repeat the time-jitter measurement. (When we get to the ADC lab in this class we will use Timer triggered ADC sampling which will completely remove the time jitter caused by higher priority interrupts).

*Deliverable*: Measure time jitter with two or more interrupts active. Try to generalize the results deriving a theoretical estimate of the time jitter of the periodic ADC sampling using software triggering.

**E. ADC noise measurements.**
Create a constant voltage of any value greater than 0 and less than 3.3V. You can use a shunt diode, two resistors between +3.3 and ground, or a battery. It is ok if the voltage is noisy because we are studying noise. However we do wish for the average voltage to be constant, not drifting up or down. Before connecting to the microcontroller use the DVM to measure the DC and AC voltages of this constant. DVMs use RMS to measure AC voltage so the AC voltage is a standard way to measure noise. The less noise in the constant the more this section will evaluate ADC noise. On the other hand, the more noise you have the more dramatic your data will look.

Connect the constant voltage to the ADC input. If you debug your software in the simulator, you should see all ADC data values the same. So debug this part on the real board. Even though Figure 2.3 was taken with 64-point hardware averaging, do not activate averaging for this part. Use the software in preparations 4 and 6 to create the probability mass function for the ADC noise. You will have to run it a few times to see typical data so you can set the range of the PMF x-axis. You can either plot the PMF data on the ST7735 LCD using any of the LCD plotting routines, or you can send the data from the LaunchPad to the PC and plot the PMF in MatLab or Excel. If you plot the data on the ST7735 use a camera to capture the plot into your lab report. How would you describe the shape of the noise process? When you run it over and over do you get the same shape of the PMF? *Hint: you should not get the same PMF because the noise is not stationary.*

**F. Central Limit Theorem.**
Look up the ADC Sample Averaging Control (**`ADC0_SAC_R`**) register in the Chapter 13 of the data sheet.

*Deliverable*: Plot PMF for hardware averaging of none, 4x, 16x, and 64x. In each case the sampling rate is fixed and there are 1000 data points used to plot the PMF function. Describe qualitatively the effect of hardware averaging on the noise process. Consider two issues 1) the shape of the PMF and 2) the signal to noise ratio. *Hint: CTL.*

*Deliverable*: Use the logic analyzer or scope to determine the effect of hardware averaging on the time to execute the ISR. Why is the thread profile like Figure 2.1 very different with hardware averaging?

**Fun activity.** Noise can vary, so before you generalize from the data you collected in this lab, go around the lab room and look at the data from other groups.

**G. Write a C function that draws lines on the ST7735:** Look ahead to Lab 3 to see how you will use this function. Basically your Lab 3 clock will call the line draw function twice, once for the hour hand and once for the minute hand. You are free to design the interface however you wish, but it should work similar to this prototype

```
//************ ST7735_Line*****************************************
//  Draws one line on the ST7735 color LCD
//  Inputs: (x1,y1) is the start point
//          (x2,y2) is the end point
// x1,x2 are horizontal positions, columns from the left edge
//               must be less than 128
//               0 is on the left, 126 is near the right
// y1,y2 are vertical positions, rows from the top edge
//               must be less than 160
//               159 is near the wires, 0 is the side opposite the wires
//        color 16-bit color, which can be produced by ST7735_Color565()
// Output: none
void ST7735_Line(uint16_t x1, uint16_t y1, uint16_t x2, uint16_t y2,
                 uint16_t color);
```

**Deliverables (exact components of the lab report)**
A) Objectives (1/2 page maximum). Simply repeat the items shown in the **Goals** section
B) Hardware Design (none for this lab)
C) Software Design (the final solution with data/time dump, time jitter and PMF calculations). **Part G)** will be included as part of Lab 3)
D) Measurement Data (you may take photographs of the scope/logic analyzer/LCD or download the data)

**Prep part 2)** Show your answers to the five questions a – f.
**Part A)** Debugging profile with scope
**Part B)** Debugging profile with logic analyzer, estimation of percentage time in main/ISR
**Part C)** Explain the critical section and present alternate solutions to removing it
**Part D)** Time-jitter measurements and generalization of factors that contribute to jitter
**Part E and F)** PMF data and discussion of results. Does your data support CLT? If not why?
**Part F)** Debugging profile of execution time in ISR with hardware averaging. Why is it different?

**Analysis and Discussion (give short 1 or two sentence answers to these questions)**
1) The ISR toggles PF2 three times. Is this debugging intrusive, nonintrusive or minimally intrusive? Justify your answer.
2) In this lab we dumped strategic information into arrays and processed the arrays later. Notice this approach gives us similar information we could have generated with a printf statement. In ways are printf statements better than dumps? In what ways are dumps better than printf statements?
3) What are the necessary conditions for a critical section to occur? In other words, what type of software activities might result in a critical section?
4) Define "minimally intrusive".
5) The PMF results should show hardware averaging is less noisy than not averaging. If it is so good why don't we always use it?

**Checkout**
You should be able to demonstrate:
Your understanding of the scope features listed in Part A and B.
Part D. Time jitter measurements (ok to demo this in the debugger, looking at a watch window).
Part E, F. Show some PMF plots with and without hardware averaging.
**Part G) Demonstrate the functions that draw lines on the LCD**
**No specific software will be turned in for this lab** (**Part G)** will be included as part of Lab 3)

There is an old Lab2 report posted on the web. This is to give you an example of how to write an EE445L lab report. As you can see from the 2007 date, this particular report was generated for different assignment from your Lab 2. I.e., look at the style, but not the content of this report.
The underlined sections identify components that must be performed and included in the lab report.

**Hints:**
1) This program makes Timer1 count down continuously at 80 MHz

```
void Timer1_Init(void){
  volatile uint32_t delay;
  SYSCTL_RCGCTIMER_R |= 0x02;   // 0) activate TIMER1
  delay = SYSCTL_RCGCTIMER_R;   // allow time to finish activating
  TIMER1_CTL_R = 0x00000000;    // 1) disable TIMER1A during setup
  TIMER1_CFG_R = 0x00000000;    // 2) configure for 32-bit mode
  TIMER1_TAMR_R = 0x00000002;   // 3) configure for periodic mode, down-count
  TIMER1_TAILR_R = 0xFFFFFFFF;  // 4) reload value
  TIMER1_TAPR_R = 0;            // 5) bus clock resolution
  TIMER1_CTL_R = 0x00000001;    // 10) enable
TIMER1A
}
```