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 © 2014.

Goals
- To develop software debugging techniques,
  - Performance debugging (dynamic or real time)
  - Profiling (detection and visualization of program activity)
- To pass data using a FIFO queue,
- To learn how to use the oscilloscope and logic analyzer,
- To observe critical sections,
- Get an early start on Lab 3, by writing a line drawing function.

Review
- Valvano Section 3.7 on FIFO queues,
- Valvano Section 3.9 on debugging,
- Valvano Section 5.3 on critical sections,
- Valvano Section 5.7 on periodic SysTick interrupts,
- The GPIO chapter of the data sheet of your microcontroller,
- Sections 3.3 and 3.4 on SysTick and NVIC of the microcontroller data sheet,
- Logic analyzer instructions.

Starter files
- **`Fifo_xxx.zip Performance_xxx.zip BadFifo.c`** (xxx is 4C123 or 4C1294)

Background

Every programmer is faced with the need to debug and verify the correctness of his or her software. In this lab, we will study hardware-level techniques like the oscilloscope and logic analyzer; and software-level tools like simulators, monitors, and profilers. **Nonintrusiveness** is the characteristic or quality of a debugger that allows the software/hardware system to operate normally as if the debugger did not exist. **Intrusiveness** is used as a measure of the degree of perturbation caused in program performance by the debugging instrument itself. For example, a **`printf`** statement added to your source code is very intrusive because it significantly affects the real-time interaction of the hardware and software. A debugging instrument is classified as **minimally intrusive** if it has a negligible effect on the system being debugged. More specifically, we will classify an instrument as minimally intrusive if the time to execute the instrument is small compared to the time interval between calls. In other words, minimally intrusive debugging code consumes a small percentage of the available processor cycles. In a real microcomputer system, breakpoints and single-stepping are also intrusive, because the real hardware continues to change while the software has stopped. When a program interacts with real-time events, the performance can be significantly altered when using intrusive debugging tools. On the other hand, dumps, dumps with filter and monitors (e.g., output strategic information on LEDs) are much less intrusive. A logic analyzer that passively monitors the activity of the software by observing existing I/O signals is completely nonintrusive. We can add extra outputs not part of the system requirements, toggle them for during execution, and observe these extra outputs on a logic analyzer. Usually we classify this process as minimally intrusive. An in-circuit emulator is nonintrusive because the software input/output relationships will be the same with and without the debugging tool.

In this lab, we will investigate if various implementations of a FIFO queue have critical sections. The particular scenario we will study is a single background task that puts, coupled with the foreground task that gets. The scenario typifies an input device or a data acquisition system. It is easy to proof a critical exists because we can devise an experiment that records the effect of the critical section: input gets lost, data is incorrect, or data arrives out of order. From a theoretical standpoint, we can eliminate critical sections by removing one of the three necessary components: writes to globals, shared globals, and nonatomic sequence. For a FIFO queue that must have writes to shared globals, we could guarantee no critical sections by disabling or disarming interrupts while performing the get in the foreground. Postponing interrupts incurs a latency cost and should be avoided if possible.

The testing approach applies these principles: stabilization, acceleration, variation, coverage, and exhaustion. **Stabilization** means we will configure the test so it focuses just on the FIFO and fixes all input parameters. This allows us to run the test multiple times and achieve the same results. In other words, we will control all the test conditions. **Acceleration** means we will speed up the process. Since we are just testing the FIFO we will increase the put and get rates in an attempt to expose possible faults. Since we can control the test environment, we will vary the time between puts and the number of elements in the FIFO. If we consider a FIFO that can hold **n** elements with the get function requiring **m** assembly language instructions to execute, there are **n*m** possible states the get function might be in when

the interrupt is requested and the put function is called. Some of the FIFO implementations have even more possibilities. For example, with the pointer-FIFO with 16 elements, the **PutPt** can be one of 16 values and the **GetPt** can also be one of 16 values. In this case, there are 16*16***m** possible states. **Coverage** defines the subset of possible states selected for testing. For complete coverage, we will vary the test conditions in order to hit each of these possible states at least once. **Exhaustion** means we will run it long enough to cover all potential conditions.

Often on an embedded system with limited debugging facilities, initially we define strategic variables as global, when proper software principles dictate they should be local. We define them as global to simplify the debugging procedures. Simple debuggers allow us to observe global variables during execution. Once the system is debugged, we can redefine them as local.

Required Hardware

EK-TM4C123GXL or EK-TM4C1294XL, www.ti.com $12.99 or 19.99
Sitronix ST7735 Color LCD, http://www.adafruit.com/products/358 $19.99

Preparation (do this before lab starts)

1. Download the FIFO_xxx project. Open the map file and record where in memory are **RxPutPt** and **RxGetPt**.
2. Look at the following C code and associated assembly created by the compiler. Answer the following questions

a) If you look in the main program of the FIFO_1968 project you will see that main calls **RxFifo_Get**. How is the **datapt** parameter passed? Is it an input or output parameter? Is it call by value or call by reference? At the time of the call where is the **datapt** located?

b) What 32-bit value is at ROM location 0x00000A34? Is it big endian or little endian? What is in R0 after the first LDR is executed? What is in R0 after the second LDR is executed?

c) What is the difference between **MOV** and **MOVS**?

d) What is the difference between **LDR** and **LDRB**?

e) What is the difference between **LDR** and **STR**?

f) Which of the assembly instructions is the return from subroutine instruction? How is the return address stored?

g) How is the return parameter passed? At the time of the return, where is the parameter?

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.

```
0x000009C4 4601    MOV  r1,r0
0x000009C6 481D    LDR  r0,[pc,#116]  ; @0x0A3C
0x000009C8 6800    LDR  r0,[r0,#0x00]
0x000009CA 4A1B    LDR  r2,[pc,#108]  ; @0x0A38
0x000009CC 6812    LDR  r2,[r2,#0x00]
0x000009CE 4290    CMP  r0,r2
0x000009D0 D101    BNE  0x000009D6
0x000009D2 2000    MOVS r0,#0x00
0x000009D4 4770    BX   lr
0x000009D6 4818    LDR  r0,[pc,#96]  ; @0x0A38
0x000009D8 6800    LDR  r0,[r0,#0x00]
0x000009DA 7800    LDRB r0,[r0,#0x00]
0x000009DC 7008    STRB r0,[r1,#0x00]
0x000009DE 4816    LDR  r0,[pc,#88]  ; @0x0A38
0x000009E0 6800    LDR  r0,[r0,#0x00]
0x000009E2 1C40    ADDS r0,r0,#1
0x000009E4 4A14    LDR  r2,[pc,#80]  ; @0x0A38
0x000009E6 6010    STR  r0,[r2,#0x00]
0x000009E8 4610    MOV  r0,r2
0x000009EA 6802    LDR  r2,[r0,#0x00]
0x000009EC 4811    LDR  r0,[pc,#68]  ; @0x0A34
0x000009EE 3020    ADDS r0,r0,#0x20
0x000009F0 4282    CMP  r2,r0
0x000009F2 D102    BNE  0x000009FA
0x000009F4 3820    SUBS r0,r0,#0x20
0x000009F6 4A10    LDR  r2,[pc,#64]  ; @0x0A38
0x000009F8 6010    STR  r0,[r2,#0x00]
```

```
// Two-pointer implementation of FIFO
// can hold 0 to RXFIFOSIZE-1 elements
#define RXFIFOSIZE 32 // can be any size
#define RXFIFOSUCCESS 1
#define RXFIFOFAIL    0

typedef uint8_t rxDataType;

rxDataType volatile *RxPutPt; // put next
rxDataType volatile *RxGetPt; // get next
rxDataType static RxFifo[RXFIFOSIZE];

int RxFifo_Get(rxDataType *datapt){
  if(RxPutPt == RxGetPt ){
    return(RXFIFOFAIL);  // Empty
  }
  *datapt = *(RxGetPt++);
  if(RxGetPt == &RxFifo[RXFIFOSIZE]){
     RxGetPt = &RxFifo[0];   // wrap
  }
  return(RXFIFOSUCCESS);
}
```

```
0x000009FA 2001   MOVS r0,#0x01
0x000009FC E7EA   B    0x000009D4

0x00000A34 00FC   LSLS r4,r7,#3
0x00000A36 2000   MOVS r0,#0x00
0x00000A38 0034   MOVS r4,r6
0x00000A3A 2000   MOVS r0,#0x00
0x00000A3C 0030   MOVS r0,r6
0x00000A3E 2000   MOVS r0,#0x00
```

3. Look at the implementations of **StartCritical** and **EndCritical** in **startup.s.** Why are these functions added to **RxFifo_Init**? **RxFifo_Init** is called from main. When this function returns are interrupts enabled or disabled? Why?

4. In this part, we will study the execution speed of the routine **Fifo_Get** the hard way. Look at the above assembly listing. Copy paste the assembly into your lab report. Assuming the FIFO is not full and the pointer does not need to wrap, highlight the assembly instructions that will be executed. Look up the execution time for each instruction in **CortexM4_TRM_r0p1.pdf** manual, section 3.3. Add up the cycle counts for each instruction, and record the sum. Use this sum to estimate the total time in µs required to call and execute the **Fifo_Get** function. For most labs this semester we will be running at 80 MHz. When you will get to a conditional branch, you need to make assumptions about which way execution will go. I.e., assume the FIFO is not empty and does not need to wrap the pointer. The assembly generated by the compiler depends on the compiler version and optimization settings. You are allowed to solve the preparation with either the assembly listed above or the assembly you see on your computer. There is not one correct solution, so expect your answer to this question to be different than your classmates.

Procedure (do this during lab)

A. Learning 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.

B. Observe the debugging profile

Connect PF3 and PF1 to the dual channel scope, run the FIFO starter project. Capture a graph that includes at least two successive interrupts. In three or four sentences describe the timing relationship between timer interrupts and execution of the foreground loop. (PN3 and PN1 on TM4C1294) In this example:

The rising edge of PF1 means start of foreground calling **RxFifo_Get**
The falling edge of PF1 means end of foreground calling **RxFifo_Get**
The rising edge of PF3 means start of interrupt
The falling edge of PF3 means end of interrupt

Does the call to **RxFifo_Put** ever occur during a call to **RxFifo_Get**?

C. Observe the debugging profile

Connect PF3, PF2, and PF1 to the logic analyzer, run the FIFO starter project. Capture a graph that includes at least two successive interrupts. In three or four sentences describe its behavior (in particular, describe the timing relationship between timer interrupts and execution of the foreground loop). Include at least two successive interrupts. (PN3, PN2, and PN1 on TM4C1294) In this example:

The rising edge of PF1 means start of foreground calling RxFifo_Get
The falling edge of PF1 means end of foreground calling RxFifo_Get
The rising edge of PF2 means start of foreground performing non-FIFO operations
The falling edge of PF2 means end of foreground performing non-FIFO operations
The rising edge of PF3 means start of interrupt
The falling edge of PF3 means end of interrupt

Use the logic analyzer data to estimate the probability that a call to **RxFifo_Put** occurs during a call to **RxFifo_Get**?

D. Instrumentation measuring with an independent counter

In the preparation, you estimated the execution speed of the `RxFifo_Get` routine by counting instruction cycles. This is a tedious, an inaccurate technique on a computer like the Arm Cortex M4. It is inaccurate because of the optimizations and parallel executions. Cycle counting can't be used in situations where the execution speed depends on external device timing (e.g., think about how long it would take to execute `UART_InChar`.) On more complex computers, there are many unpredictable factors that can affect the time it takes to execute single instructions, many of which can't be predicted *a priori*. Some of these factors include an instruction cache, out of order instruction execution, branch prediction, data cache, virtual memory, dynamic RAM refresh, DMA accesses, data synchronization between multiple buses, and coprocessor operation. For systems with these types of activities, it is not possible to predict execution speed simply by counting cycles using the processor data sheet. Luckily, most computers have a timer that operates independently from these activities. In the Arm Cortex M, there is a 24-bit counter called SysTick that is decremented every bus clock. It automatically rolls over when it gets to 0. If we are sure the execution speed of our function is less than (2^{24} counts), we can use this timer to directly measure execution speed with only a modest amount of intrusiveness. Download the **Performance_xxx** project. Make a copy of your Lab 1 project and include the FIFO and SysTick modules into this new project, so you have FIFO, SysTick, and printf functionality running at 80 MHz. (120MHz on the TM4C1294). Use something like the following main program to measure the execution time of `RxFifo_Get`.

```
volatile unsigned uint32_t before, elapsed;
int main(void){ rxDataType data; int volatile result;
  PLL_Init();        // 80 MHz
  SysTick_Init(); // initialize SysTick timer, see SysTick.c
  RxFifo_Init();
  Output_Init();
  Output_Color(ST7735_BLUE);
  RxFifo_Put(12);
  before = NVIC_ST_CURRENT_R;
  result = RxFifo_Get(&data);
  elapsed = (before-NVIC_ST_CURRENT_R)&0x00FFFFFF;
  printf("Time is %u bus cycles.\n", elapsed);
  while(1){};
}
```

Measure the execution speeds of these three get functions.

```
// Version 1) with no debugging
int RxFifo_Get(rxDataType *datapt){
  if(RxPutPt == RxGetPt ){
    return(RXFIFOFAIL);  // Empty
  }
  *datapt = *(RxGetPt++);
  if(RxGetPt == &RxFifo[RXFIFOSIZE]){
     RxGetPt = &RxFifo[0];   // wrap
  }
  return(RXFIFOSUCCESS);
}
```

Program 2.1. Original FIFO function.

```
// Version 2) with debugging print
int RxFifo_Get2(rxDataType *datapt){
  if(RxPutPt == RxGetPt ){
    return(RXFIFOFAIL);  // Empty
  }
```

```
  *datapt = *(RxGetPt++);
  if(RxGetPt == &RxFifo[RXFIFOSIZE]){
     RxGetPt = &RxFifo[0];   // wrap
  }
  printf("RxGetPt = %x , data= %d\n", RxGetPt, *datapt);
  return(RXFIFOSUCCESS);
}
```

Program 2.2. Using printf output to debug the FIFO function.

```
// Version 3) with debugging dump
uint32_t ptBuf[10];
rxDataType dataBuf[10];
uint32_t Debug_n=0;
int RxFifo_Get3(rxDataType *datapt){
  if(RxPutPt == RxGetPt ){
    return(RXFIFOFAIL);  // Empty
  }
  *datapt = *(RxGetPt++);
  if(RxGetPt == &RxFifo[RXFIFOSIZE]){
     RxGetPt = &RxFifo[0];   // wrap
  }
  if(Debug_n<10){
    ptBuf[Debug_n] = (uint32_t) RxGetPt;
    dataBuf[Debug_n] = *datapt;
    Debug_n++;
  }
  return(RXFIFOSUCCESS);
}
```

Program 2.3. Using a dump to debug the FIFO function.

Collect execution times for the function **version 1)** as is, **version 2)** with debugging print statements, and **version 3)** with debugging dump statements. The output of version 1 should be similar to the execution speed you calculated by cycle-counting as part of the preparation. For the dump case, you are measuring the time to store into the array and not the time to print the array on the screen. The slow-down introduced by the debugging procedures defines its level of intrusiveness.

E. Instrumentation Output Port.

Another method to measure real time execution involves an output port and an oscilloscope. Connect Port F bit 0 to an oscilloscope. You will create a debugging instrument that sets Port F bit 2 to one just before calling **`RxFifo_Get`**. Then, you will set the output back to zero right after. You will set the port's direction register to 1, making it an output. If you were to put the instruments inside **`RxFifo_Get`**, then you would be measuring the speed of the calculations and neglecting the time it takes to pass parameters and perform the subroutine call. On the other hand, in a complex system this method allows you to visualize each call to **`RxFifo_Get`**, regardless from where it was called. For this lab, stabilize the input, and repeat the operation in a loop, so that the scope can be triggered. The time measured in this way includes the overhead of passing parameters. E.g.,

```
int main(void){ int result;
rxDataType data;
  PLL_Init();                       // bus clock at 80 MHz
  RxFifo_Init();
  SYSCTL_RCGCGPIO_R |= 0x20;        // activate port F
  while((SYSCTL_PRGPIO_R&0x20)==0){};
  GPIO_PORTF_DIR_R |= 0x0E;         // make PF3-1 output (PF3-1 built-in LEDs)
  GPIO_PORTF_AFSEL_R &= ~0x0E;      // disable alt funct on PF3-1
  GPIO_PORTF_DEN_R |= 0x0E;         // enable digital I/O on PF3-1
  GPIO_PORTF_PCTL_R = (GPIO_PORTF_PCTL_R&0xFFFF000F)+0x00000000; // GPIO
```

```
  GPIO_PORTF_AMSEL_R = 0;          // disable analog functionality on PF
  for(;;){
    RxFifo_Put(1);
    PF2 = 0x04;
    result = RxFifo_Get(&data);
    PF2 = 0;
  }
}
```

Compare the results of this measurement to the **SysTick** method. Discuss the advantages and disadvantages between the **SysTick** and scope techniques. Determine the measurement error of the scope technique using the following code. The time the signal is high represents an offset error introduced by the measurement itself.

```
  for(;;){
    PF2 = 0x04;
    PF2 = 0;
  }
```

F. Proving there are no critical sections in RxFifo

Write access to shared global variables sometimes create critical sections. The data passed from background to foreground is an incremental sequence (i.e., 0, 1, 2, … 255, 0, 1, 2 …). The consequence of a critical section is typically corrupted data (lost data, changed data, or extra data). Run the system with the **RxFifo** observing the **Histogram** and **Errors**. Run until each instruction in the get function is interrupted at least once. Because of the optimization in the Arm Cortex M, the processor may not interrupt between a store instruction (**STR**) and a simple register move instruction (**MOV**). You are free to use any of the debugging techniques in this lab, but collect measurement data demonstrating that this FIFO has no critical sections.

G. Proving there are critical sections in this BadFifo

Run the system with the **BadFifo** observing the **Histogram** and **Errors**. While running with the bad FIFO, use debugging skills to determine what happened just prior to an error. You are free to use any of the debugging techniques in this lab, but collect measurement data specifying the sequence of events that resulted in an error.

H. Thread Profile using a logic analyzer.

Replace the FIFO calls with calls to **TxFifo** (the index FIFO). Run the critical section test (F) to verify this FIFO also has no critical sections. Remove the Port F outputs from the main program. When the execution pattern is very complex, you could use a hardware technique to visualize which program is currently running. In this section, choose the timer interrupt service routine and at least three other regular functions to profile. You will associate one output pin (e.g., PF3, PF2, PF1, PF0) with each function you are profiling. You will connect all the output pins to the logic analyzer to visualize in real time the function that is currently running. For each regular routine, set its output bit high when you start execution and clear it low when the function completes. E.g., assume PF1 and PF3 are associated with **TxFifo_Get** and **TxFifo_Put**.

```
int TxFifo_Get(txDataType *datapt){
  PF1 = 0x02;
  if(TxPutI == TxGetI ){
    PF1 = 0x00;
    return(TXFIFOFAIL); // Empty
  }
  *datapt = TxFifo[TxGetI&(TXFIFOSIZE-1)];
  TxGetI++;  // Success, update
  PF1 = 0x00;
  return(TXFIFOSUCCESS);
}
```

```
int TxFifo_Put(txDataType data){
  PF3 = 0x08;
  if((TxPutI-TxGetI) & ~(TXFIFOSIZE-1)){
    PF3 = 0x00;
    return(TXFIFOFAIL); // fifo full
  }
  TxFifo[TxPutI&(TXFIFOSIZE-1)] = data;
  TxPutI++;  // Success, update
  PF3 = 0x00;
  return(TXFIFOSUCCESS);
}
```

Compile, download, and run this system observing on the logic analyzer the behavior of the four output pins. Explain what is happening. If the data you collect is confusing, change which functions you are profiling and repeat the profiling.

I. Write a C function that draws lines on the ST7735: Look ahead to Lab 3 to see how you will use these functions. 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

```
//************ 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 (no software printout in the report, **Part I)** will be included as part of Lab 3)

D) Measurement Data (you may sketch the waveforms or use the printer connection)

Prep part 4) Show the cycle counting of execution speed. Include the listing file, circling or highlighting the instructions used to determine execution speed. Include the execution speed in cycles.

Parts B and C) Sketch and describe the profiles, showing a situation occurring with two successive interrupts

Part D) Show the three results of the execution times

Part E) Show the execution time measured with the oscilloscope, discuss advantages of two methods

Part F and H) Give convincing data showing that RxFifo and TxFifo have no critical sections

Part G) Debugging data showing the sequence of execution that causes FIFO data to be corrupted.

Part H) Show the thread profile measured with the logic analyzer showing at least two interrupts

E) Analysis and Discussion (give short 1 or two sentence answers to these questions)

1) You measured the execution speed of **`RxFifo_Get`** three ways. Did you get the same result? If not, explain.

2) Which method of measuring execution speed would you use if you expected the execution speed to vary a lot (e.g., ranging from 0.5 to 20ms), and wanted to determine the minimum, maximum and average speed? Why?

3) Which method of measuring execution speed would you use if you expected the execution speed to be very large (e.g., 20 seconds)? Why?

4) Define "minimally intrusive".

5) List the two necessary components collected during a "profile".

6) What is the critical section in the bad FIFO? How would you remove the critical section?

Checkout

You should be able to demonstrate:

Your understanding of the scope features listed in Part A.

Part E. Instrumentation output port.

Part F,G. Show how you observed the critical section.

Part H. Profiling using an output port.

Part I) Demonstrate the functions that draw lines on the LCD

No specific software will be turned in for this lab (Part I) 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.