Lab 2g Performance Debugging

This laboratory assignment accompanies the book, Embedded Microcomputer Systems: Real Time Interfacing, Second edition, by Jonathan W. Valvano, published by Thomson, copyright © 2006.

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 logic analyzer,
- to observe critical sections.

Review
- Valvano Section 2.11 on debugging,
- Valvano Section 4.15.3 on periodic interrupts,
- Port T and TCNT of the 9S12DP512 Technical Data Manual,
- Output compare interrupts on the 9S12DP512 in the Technical Data Manual,
- Logic analyzer instructions.

Starter files
- **`Lab2g_DP512`** (used for most of this lab) and **`Lab2g_v2_DP512`** (used for procedure B)

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 **`SCI_OutUDec`** or **`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 CPU 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 the memory bus cycles is completely nonintrusive. An in-circuit emulator is also nonintrusive because the software input/output relationships will be the same with and without the debugging tool. Similarly, breakpoints and single-stepping on a simulator like **TExaS** are nonintrusive, because the simulated hardware and the software are affected together.

Often on an embedded system like the 9S12DP512 with limited debugging facilities, initially we define strategic variables as global (e.g., **`BackData ForeData`**), when proper software principles dictate they should be local. We define them as global to simplify the debugging procedures. The debugger we use allows us to observe global variables during execution. Once the system is debugged, we can redefine them as local. On the other hand, some variables must be defined global (e.g., **`NumLost`**), because we want the information to be permanent.

Preparation (do this before lab starts)

1. Look at the following C code and associated assembly created by the compiler. Place six circles on the listing showing: the assembly instruction that allocates **`i`**; the four assembly instructions that access **`i`**; and the assembly instruction that deallocates **`i`**. Limit your search to those instructions that access **`i`** while it is on the stack, and do not include all the places the value of **`i`** is manipulated. In particular, there should be one place where **`i`** is allocated, four places where it is accessed, and one place where it is deallocated. Using the assembly listing, how long does it take to execute **`PTT_PTT0 = 1;`**

```
Function: OC0Han
  0000 36       [2]   PSHA
  0001 c604     [1]   LDAB  #4
  0003 87       [1]   CLRA
  0004 0700     [4]   BSR   Debug_Profile
  0006 1c000001 [4]   BSET  _PTT,#1
  000a c601     [1]   LDAB  #1
  000c 5b00     [2]   STAB  _TFLG1
  000e dc00     [3]   LDD   _TC0
  0010 c35dc0   [2]   ADDD  #24000
  0013 5c00     [2]   STD   _TC0
  0015 6980     [2]   CLR   0,SP
  0017 2011     [3]   BRA   *+19 ; 002a
  0019 e680     [3]   LDAB  0,SP
  001b 160000   [4]   JSR   RxFifo_Put
  001e 046407   [3]   TBNE  D,*+10 ; 0028
  0021 fe0000   [3]   LDX   NumLost
  0024 08       [1]   INX
  0025 7e0000   [3]   STX   NumLost
  0028 6280     [3]   INC   0,SP
  002a f60000   [3]   LDAB  BackData
  002d e180     [3]   CMPB  0,SP
  002f 24e8    [3/1]  BCC   *-22 ; 0019
  0031 720000   [4]   INC   BackData
  0034 f60000   [3]   LDAB  BackData
  0037 c10a     [1]   CMPB  #10
  0039 2603    [3/1]  BNE   *+5 ; 003e
  003b 790000   [3]   CLR   BackData
  003e 1d000001 [4]   BCLR  _PTT,#1
  0042 32       [3]   PULA
  0043 0b       [8]   RTI
```

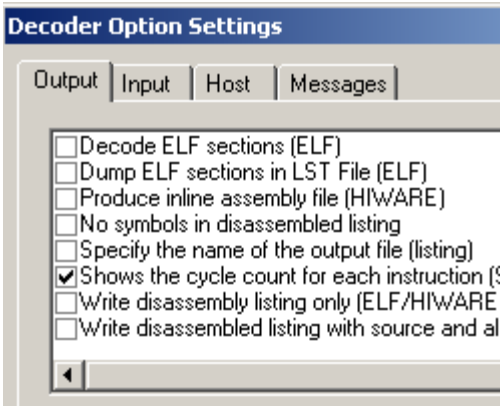
```
interrupt 8 void OC0Han(void){
unsigned char i;
  Debug_Profile(4);
  PTT_PTT0 = 1;
  TFLG1 = 0x01;      // ack
  TC0 = TC0 +24000;
  for(i=0; i<=BackData; i++){
    if(RxFifo_Put(i)==0){
      NumLost++;
    }
  }
  BackData++;
  if(BackData==10){
    BackData = 0;   // 0 to 9
  }
  PTT_PTT0 = 0;
}
```

2. Compile the Lab2g project and look at the associated map file. Print out that portion of the map file showing where the variables **`BackData ForeData NumLost`** are stored in memory. Do not print the entire map file, just the one page containing the addresses of these three variables. We will be using the listing and map files to debug our C programs.

3. In this part, we will study the execution speed of the routine **`Fifo_Get`** the hard way. Open the assembly listing file showing **`Fifo_Get`** (**`FIFO.lst`**). Edit the assembly listing files leaving just the assembly code that implements **`Fifo_Get`**. Include also the assembly code from the main program that calls **`Fifo_Get`** (**`Lab2g.lst`**). Print out this subset of the assembly listings. Please don't print the entire listing files, just the parts that implement this function and the call to this function. Add up the cycle counts for each instruction, and record the sum on the printout. Use this sum to estimate the total time in µs required to call and execute the **`Fifo_Get`** function. In load mode, the 9S12DP512 runs at 24 MHz. In run mode, the 9S12DP512 begins execution at 8 MHz. Because we add code to initialize the phase-lock-loop (PLL), we will be running at 24 MHz in both modes. 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).

This section was a little easier because we set up the compiler to place the cycle counts in the assembly listing. In particular,

execute **Edit->MonitorSettings…**
click on **Importer for HC12**
click the **Options** button
select the **Output** tab
select the **Shows the cycle count for each instruction**



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; and 7) 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 PT1 and PT0 to the dual channel scope, run the Lab2g starter project and describe its behavior (in particular, describe the timing relationship between output compare interrupts and execution of the foreground loop). In this example:

the falling edge of PT1 means start of foreground waiting
the rising edge of PT1 means start of foreground processing
the rising edge of PT0 means start of interrupt
the falling edge of PT0 means end of interrupt

C. Instrumentation measuring with an independent counter, `TCNT`

In the preparation, you estimated the execution speed of the **`Fifo_Get`** routine by counting instruction cycles. This is a tedious, but accurate technique on a computer like the 9S12 (when running in single chip mode). It is accurate because each instruction (e.g., **`LDAA B,X`**) always executes in exactly the same amount of time. 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 **`SCI_InChar`**.) If the 9S12 were to be running in expanded mode, the time for each instruction would depend also on whether it is accessing internal or external memory. 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 9S12, there is a 16-bit counter called **`TCNT`** that is incremented every E clock. The 9S12DP512 has a prescaler that can be placed between the E clock and the **`TCNT`** counter. Leave the prescaler at its default value of divide by 1 (i.e., when **`TSCR2`** is 0, **`TCNT`** incremented each bus cycle). It automatically rolls over when it gets to $FFFF. If we are sure the execution speed of our function is less than (65535 counts), we can use this timer to directly measure execution speed with only a modest amount of intrusiveness. Let **`First`** and **`Delay`** be unsigned 16-bit **global** integers. The following code will set the variable **`Delay`** to the execution speed of **`Fifo_Get`**.

```
TSCR1 = 0x80;     // Enable TCNT, 24MHz assuming PLL is active
Fifo_Init();    // Initialize fifo
Fifo_Put(1);    // make sure there is something in the fifo
First = TCNT;
Fifo_Get(&ForeData);
Delay = TCNT-First-6;
for(;;){}
```

The constant "6" is selected to account for the time overhead in the measurement itself. In particular, run the following,

```
First = TCNT;
Delay = TCNT-First-6;
```

and adjust the "6" so that the result calculated in **`Delay`** is zero.

```
// Version 1) with no debugging
int Fifo_Get(dataType *datapt){
  if(PutI == GetI ){
    return(FIFOFAIL); // Empty if PutI=GetI
  }
  *datapt = Fifo[GetI&(FIFOSIZE-1)];
  GetI++;  // Success, update
  return(FIFOSUCCESS);
}
```

Program 2.1. Original FIFO function.

```
// Version 2) with debugging print
int Fifo_Get2(dataType *datapt){
  if(PutI == GetI ){
    return(FIFOFAIL); // Empty if PutI=GetI
  }
  *datapt = Fifo[GetI&(FIFOSIZE-1)];
  SCI0_OutUHex(GetI&(FIFOSIZE-1));
  SCI0_OutChar(32);
  SCI0_OutUDec((unsigned short)*datapt);
  SCI0_OutChar(13); SCI0_OutChar(10);
  GetI++;  // Success, update
  return(FIFOSUCCESS);
}
```

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

```
// Version 3) with debugging dump
unsigned short ptBuf[10];
dataType dataBuf[10];
unsigned short Debug_n=0;
int Fifo_Get3(dataType *datapt){
  if(PutI == GetI ){
    return(FIFOFAIL); // Empty if PutI=GetI
  }
  *datapt = Fifo[GetI&(FIFOSIZE-1)];
  if(Debug_n<10){
    ptBuf[Debug_n] =  GetI&(FIFOSIZE-1);
    dataBuf[Debug_n] = *datapt;
    Debug_n++;
  }
  GetI++;  // Success, update
  return(FIFOSUCCESS);
}
```

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. For this section, you will use the starter files in **`Lab2g_v2_DP512`**, which include the SCI.C and SCI.H files from the SCI project (and adjust the baud rate initialization to account for the 24 MHz clock.) 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.

D. Instrumentation Output Port.

Another method to measure real time execution involves an output port and an oscilloscope. For this section, you will use the **`main2`** program in **`Lab2g_DP512`**. Connect Port T bit 0 to an oscilloscope. You will create a debugging instrument that sets Port T bit 0 to one just before calling **`Fifo_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 **`Fifo_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 **`Fifo_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.,

```
void main(void){
char data;
  DDRT |= 0x01;
  Fifo_Init();  // initialize
asm cli
  for(;;){
    Fifo_Put(1);
    PTT_PTT0 = 1;
    Fifo_Get(&data);
    PTT_PTT0 = 0;
  }
}
```

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

```
void main(void){
  DDRT |= 0x01;
  for(;;){
    PTT_PTT0 = 1;
    PTT_PTT0 = 0;
  }
}
```

E. Profiling using a software dump to study execution pattern

The objective of this part is to develop software and hardware techniques to visualize program execution in real time. For this section, you will use the starter files in **Lab2g_DP512**. This system has two threads: the foreground thread (**main** program) and a background thread (output compare interrupt handler). The background thread, invoked by the output compare clock hardware, executes periodically. The foreground thread runs in the remaining intervals. The background thread calls **Fifo_Put** and the foreground thread calls **Fifo_Get**. In this way data is passed between the threads. In this lab, we will study the execution pattern of this two-threaded system that uses the FIFO. E.g.,

```
unsigned short timeBuf[100];
unsigned short placeBuf[100];
unsigned short Debug_n=0;
void Debug_Profile(unsigned short thePlace){
  if(Debug_n>99) return;
  timeBuf[Debug_n] = TCNT;        // record current time
  placeBuf[Debug_n] = thePlace;   // record place from which it is called
  Debug_n++;
}
```

Calls to **Debug_Profile** have been placed at strategic places in the software system. The **thePlace** parameter specifies where the software is executing. The **timeBuf** records when the debugging profile was called. Notice that the debugging instrument saves in an array (this is a dump). In this experiment, we will first run the 9S12 filling **timeBuf** and **placeBuf**, then use the debugger to obverse the buffers. Run the instrumented system and make a hardcopy printout the results of the debugging instruments. Assume the **BackData** is 3. On the source code draw "tail to head" arrows (e.g.,→) illustrating the execution pattern starting the output compare ISR and ending when all the data has been processed through the system. The picture should look like Figures 2.36 and 2.37 on page 127 of the book. Consider the two threads that exist in this system: the background ISR and the foreground main program. Draw a data-flow graph of this system showing connection between the two threads. Label each arrow with the average bandwidth. If the data you collect is confusing, repeat the experiment with more (or less) instruments.

The Metrowerks debugger allows you to observe memory while the program is executing using Periodical mode. An interesting question to ask "is the Periodical Mode on the Metrowerks debugger minimally intrusive?" It takes about 5 SCI transmissions (3 frames out and 2 frames in) for the Metrowerks debugger running on the PC to read one 16-bit value from your 9S12. At 115200 baud rate, this takes about 430 μs. Even though the serial monitor uses interrupts (that is why you have to enable interrupts even if your software doesn't use interrupts) your program will be halted for the 430 μs while this data is being transferred. If you set up a debugger window with ten 16-bit

words and ask the debugger to update this window periodically every 1 second, then this debugging monitor utilizes about 4.3ms out of every 1 second of execution time. This is why it is less intrusive to collect the data by first having the 9S12 dump into arrays without Periodical mode active; then after the arrays are full, the data is transferred from the 9S12 to the Metrowerks debugger for display.

F. Thread Profile using hardware.

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 output compare interrupt service routine and at least three other regular functions to profile. You will associate one output pin (e.g., PT3, PT2, PT1, PT0) 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 PT2 and PT3 are associated with **`Fifo_Get`** and **`Fifo_Put`**

```
int Fifo_Get(dataType *datapt){
PTT_PTT2 = 1;
  if(PutI == GetI ){
    PTT_PTT2 = 0;
    return(FIFOFAIL);
  }
  *datapt = Fifo[GetI&(FIFOSIZE-1)];
  GetI++;  // Success, update
  PTT_PTT2 = 0;
  return(FIFOSUCCESS);
}
```

```
int Fifo_Put(dataType data){
PTT_PTT3 = 1;
  if((PutI-GetI) & ~(FIFOSIZE-1)){
    PTT_PTT3 = 0;
    return(FIFOFAIL); // Failed, fifo full
  }
  Fifo[PutI&(FIFOSIZE-1)] = data; // put
  PutI++;  // Success, update
  PTT_PTT3 = 0;
  return(FIFOSUCCESS);
}
```

For output compare interrupt service routine, save the previous value, set its output bit high when you start execution and restore the previous value when the function completes. E.g., assume PT0 is associated with **`OC0Han.`** 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.

```
interrupt 8 void OC0Han(void){
unsigned short i;   char previous;
  previous = PTT;
  PTT = 0x01;
  TFLG1 = 0x01;
  TC0 = TC0+24000;
  for(i=0; i<=BackData; i++){
    if(Fifo_Put(i)==0){
      NumLost++;
    }
  }
  BackData++;
  if(BackData==10){
    BackData = 0;   // 0 to 9
  }
  PTT = previous;
}
```

G. Observing critical sections

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 good FIFO and the bad FIFO observing **`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.

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)

none

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

Prep part 3) 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.

Part B) Sketch and describe the profile, showing a situation occurring with two successive interrupts

Part C) Show the three results of the execution times

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

Part E) The software profile data, draw arrows on the listings, and data-flow graph

Part F) The hardware profile data and explanation

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

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

1) You measured the execution speed of **`Fifo_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?

Checkout

You should be able to demonstrate:

Your understanding of the scope features listed in Part A.

Part D. Instrumentation output port.

Part E. Profiling using a software dump.

Part F. Profiling using an output port.

Part G. Show how you observed the critical section.

No specific software will be turned in for this lab

There is an old Lab2 report posted on the web. This is to give you an example of how to write an EE345L 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.