

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.
- Review**
- Valvano Section 2.11 on debugging,
 - Valvano Section 4.15.3 on periodic interrupts,
 - Port T and TCNT of the 9S12C32 Technical Data Manual,
 - Output compare interrupts on the 9S12C32 in the Technical Data Manual,
 - Logic analyzer instructions.
- Starter files**
- **Lab2g** (used for most of this lab) and **Lab2g_v2** (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 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** 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. 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 **TEaS** are nonintrusive, because the simulated hardware and the software are affected together.

Often on an embedded system like the 9S12C32 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)

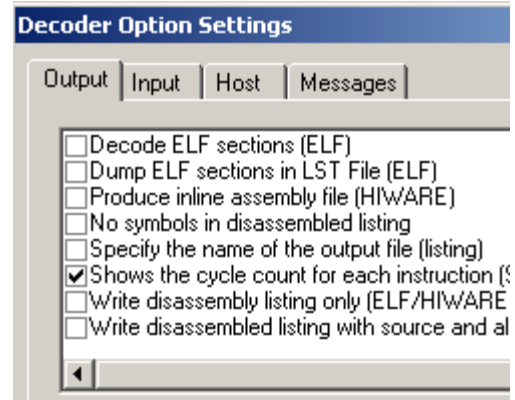
1. Copy the **Lab2g** project from class web site. Compile this system and observe the assembly listing file (e.g., **main.lst**). Print out that portion of the assembly listing that implements output compare interrupt service routine. Circle on the listing, those assembly instructions that access the local variable **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, above four places it is accessed, and one place where it is deallocated. Also look at the associated map file. Print out that portion of the map file showing where is the variables **BackData ForeData NumLost** are stored in memory. We will be using the listing and map files to debug our C programs.

2. In this part, we will study the execution speed of the routine **RxFifo_Get** the hard way. Open the assembly listing file showing **RxFifo_Get (Rx FIFO.lst)**. This time edit the assembly listing files leaving just the assembly code that implements **RxFifo_Get**. Include also the assembly code from the main program that calls **RxFifo_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. Look up the cycle counts for each instruction, and record them on the printout. Estimate the total time in μ s required to call and execute the **RxFifo_Get** function. In load mode, the 9S12C32 runs at 24MHz. In run mode, the 9S12C32 begins execution at 4 MHz. Because we add code to initialize the phase-lock-loop (PLL), we will be running at 24MHz 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 will be a little easier if you 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. Observe the debugging profile

Connect PT1 and PT0 to the dual channel scope, run the Lab2g starter project and describe its behavior.

- 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

B. Instrumentation measuring with an independent counter, TCNT

In the preparation, you estimated the execution speed of the `RxFifo_Get` routine by counting instruction cycles. This is a tedious, but accurate technique on a computer like the 6812 (when running in single chip mode). It is accurate because each instruction (e.g., `LDY 2,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 6812 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, 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 6812, there is a 16-bit counter, called `TCNT`, which is incremented every E clock. The 9S12C32 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 (`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 `RxFifo_Get`.

```

TCSCR1 = 0x80;    // Enable TCNT, 24MHz assuming PLL is active
RxFifo_Init();   // Initialize fifo
RxFifo_Put(1);   // make sure there is something in the fifo
First = TCNT;
RxFifo_Get(&ForeData);
Delay = TCNT-First-8;
for(;;){}

```

The constant "8" is selected to account for the time overhead in the measurement itself. In particular, run the following,

```

First = TCNT;
Delay = TCNT-First-8;

```

and adjust the "8" so that the result calculated in `Delay` is zero. Use this method to verify the general expression you developed as part of the preparation.

```
// Version 1) with no debugging
int RxFifo_Get(unsigned char *datapt){
    if(RxPutPt == RxGetPt){
        return(0);
    }
    else{
        *datapt = *(RxGetPt);
        RxGetPt++;
        if(RxGetPt == &RxFifo[RXFIFOSIZE]){
            RxGetPt = &RxFifo[0];
        }
        return(1);
    }
}
```

```
// Version 2) with debugging print
int RxFifo_Get(unsigned char *datapt){
    if(RxPutPt == RxGetPt){
        return(0);
    }
    else{
        *datapt = *(RxGetPt);
        SCI_OutUHex((unsigned short)RxGetPt);
        SCI_OutChar(32);
        SCI_OutUDec(*datapt);
        SCI_OutChar(13); SCI_OutChar(10);
        RxGetPt++;
        if(RxGetPt == &RxFifo[RXFIFOSIZE]){
            RxGetPt = &RxFifo[0];
        }
        return(1);
    }
}
```

```
// Version 3) with debugging dump
unsigned short ptBuf[10];
char dataBuf[10];
unsigned short Debug_n=0;
int RxFifo_Get(unsigned char *datapt){
    if(RxPutPt == RxGetPt){
        return(0);
    }
    else{
        *datapt = *(RxGetPt);
        if(Debug_n<10){
            ptBuf[Debug_n] = (unsigned short)RxGetPt;
            dataBuf[Debug_n] = *datapt;
            Debug_n++;
        }
        RxGetPt++;
        if(RxGetPt == &RxFifo[RXFIFOSIZE]){
            RxGetPt = &RxFifo[0];
        }
        return(1);
    }
}
```

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**, 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.) 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.

C. Instrumentation Output Port.

Another method to measure real time execution involves an output port and an oscilloscope. 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 **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.,

```
void main(void){
    char data;
    DDRT |= 0x01;
    RxFifo_Init(); // initialize
    for(;;){
        RxFifo_Put(1);
        PTT |= 0x01;
        RxFifo_Get(&data);
        PTT &= ~0x01;
    }
}
```

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 |= 0x01;
    PTT &= ~0x01;
  }
}
```

D. 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. 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 **RxFifo_Put** and the foreground thread calls **RxFifo_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 linked-list 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 (like a dump). The Metrowerks debugger allows you to observe memory while the program is executing. In particular use the debugger to collect the profile data. Except for a modest increase in the execution time, your instrument should not modify the operation. Run the instrumented system and make a hardcopy printout the results of the debugging instruments. On the source code listings draw "tail to head" arrows (e.g.,→) illustrating the execution pattern as one piece of data is passed through the system. Draw a data-flow graph of this system. If the data you collect is confusing, repeat the experiment with more (or less) instruments.

E. 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 PT3 is associated with **RxFifo_Put**

```
int RxFifo_Put(char data){
  char volatile *tempPt;
  PTT |= 0x08;
  tempPt = RxPutPt;
  *(tempPt) = data;          // try to Put data into fifo
  tempPt++;
  if(tempPt == &RxFifo[RXFIFOSIZE]){ // need to wrap?
    tempPt = &RxFifo[0];
  }
  if(tempPt == RxGetPt){
    PTT &= ~0x08;
    return(0);              // Failed, fifo was previously full
  }
  else{
    RxPutPt = tempPt;      // Success, so update pointer
    PTT &= ~0x08;
    return(1);
  }
}
```

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+1875;
  for(i=0; i<=BackData; i++){
    if(RxFifo_Put(i)==0){
      NumLost++;
    }
  }
  BackData++;
  if(BackData==20){
    BackData = 0; // 0 to 19
  }
  PTT = previous;
}

```

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 2) 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 A) Sketch and describe the profile, showing a situation occurring with two successive interrupts

Part B) Show the three results of the execution times

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

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

Part E) The hardware profile data and explanation

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”.

Checkout

You should be able to demonstrate:

Part C. Instrumentation output port.

Part D. Profiling using a software dump.

Part E. Profiling using an output port.

No specific software will be turned in for this lab

The underlined sections identify components that must be performed and included in the lab report.