Lab 3c Performance Debugging

This laboratory assignment accompanies the book, Embedded Microcomputer Systems: Real Time Interfacing, by Jonathan W. Valvano, published by Brooks-Cole, copyright © 2000.

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,
• Port T and TCNT of the MC68HC812A4 or 9S12C32 Technical Data Manual,
• RTI interrupts on the MC68HC812A4 or 9S12C32 in the Technical Data Manual,
• Logic analyzer instructions.

Starter files • **`Lab3`** project

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 **TExaS** are nonintrusive, because the simulated hardware and the software are affected together.

Preparation (do this before lab starts)

1. Copy the **`Lab3`** 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 RTI interrupt service routine. Circle on the listing, those assembly instructions that access the local variable **`i`**. 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. Connect PT1 and PT0 to the dual channel scope, run the program 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
3. In this part, we will study the execution speed of the routine **`LLFifo_Put`** the hard way. Open the assembly listing files showing **`LLFifo_Put`** (**`LLFIFO.lst`**) and **`Heap_Allocate`** (**`HEAP.lst`**). This time edit the assembly listing files leaving just the assembly code that implements **`LLFifo_Put`** and **`Heap_Allocate`**. Include also the assembly code from the RTI interrupt that calls **`LLFifo_Put`**. Print out this assembly listing of these two functions. Please don't print the entire listing files, just the parts that implements these two functions. Look up the cycle counts for each instruction in the two functions, and record them on the printout. Estimate the total time in µs required to call and execute the **`LLFifo_Put`** function. The MC68HC812A4 runs at 8 MHz meaning there are 125 ns/cycle. In boot mode, the 9S12C32 runs at 24MHz. In run mode, the 9S12C32 executes at 4 MHz. When you will get to a conditional branch, you need to make assumptions about which way execution will go.

Procedure (do this during lab)

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

In the preparation, you estimated the execution speed of the `LLFifo_Put` 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., `ldd 4,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., consider how long it takes 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. Both the MC68HC812A4 and the 9S12C32 have prescalers 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 `LLFifo_Put`.

```
First = TCNT;
LLFifo_Put(1);
Delay = TCNT-First-8;
```

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.

```
// with debugging print
int LLFifo_Put(unsigned short data){
NodePtr pt; int success;
  success = 0;
  pt = (NodePtr)Heap_Allocate();
  if(pt){
    pt->value = data;
    pt->NextPt = null;
    if(PutPt){
      PutPt->NextPt = pt;
    }
    else{
      GetPt = pt;
    }
    PutPt = pt;
    success = 1;
  }
  SCI_OutUHex(pt);
  SCI_OutChar(SP);
  SCI_OutUDec(pt->value);
  SCI_OutChar(CR);
  return(success);
}
```

```
// with debugging dump
unsigned short ptBuf[10];
unsigned short dataBuf[10];
unsigned short Debug_n=0;
int LLFifo_Put(unsigned short data){
NodePtr pt; int success;
  success = 0;
  pt = (NodePtr)Heap_Allocate();
  if(pt){
    pt->value = data;
    pt->NextPt = null;
    if(PutPt){
      PutPt->NextPt = pt;
    }
    else{
      GetPt = pt;
    }
    PutPt = pt;
    success = 1;
  }
  if(Debug_n<10){
    ptBuf[Debug_n] = pt;
    dataBuf[Debug_n++] = pt->value;
  }
  return(success);
}
```

Collect execution times for the function 1) as is, 2) with debugging print statements, and 3) with debugging dump statements. 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.

B. Instrumentation Output Port.

Another method to measure real time execution involves an output port and an oscilloscope. Connect PORTT bit 0 to an oscilloscope. You will create a debugging instrument that sets PORTT bit 0 to one just before calling `LLFifo_Put`. 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 `LLFifo_Put`, 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 `LLFifo_Put`, 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.,

```
// MC68HC812A4
void main(void){
unsigned short data;
  DDRT |= 0x01;
  COPCTL = 0x00;
  LLFifo_Init();  // initialize
  for(;;){
    PORTT |= 0x01;
    LLFifo_Put(1);
    PORTT &= ~0x01;
    LLFifo_Get(&data);
  }
}
```

```
// 9S12C32
void main(void){
unsigned short data;
  DDRT |= 0x01;
  LLFifo_Init();  // initialize
  for(;;){
    PTT |= 0x01;
    LLFifo_Put(1);
    PTT &= ~0x01;
    LLFifo_Get(&data);
  }
}
```

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.

```
// MC68HC812A4
void main(void){
  DDRT |= 0x01;
  COPCTL = 0x00;
  for(;;){
    PORTT |= 0x01;
    PORTT &= ~0x01;
  }
}
```

```
// 9S12C32
void main(void){
  DDRT |= 0x01;
  for(;;){
    PTT |= 0x01;
    PTT &= ~0x01;
  }
}
```

C. 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 (RTI interrupt handler). The background thread, invoked by the RTI clock hardware, executes periodically. The foreground thread runs in the remaining intervals. The background thread calls `LLFifo_Put` and the foreground thread calls `LLFifo_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 debugger for both the MC68HC812A4 and the 9C12C32 allow 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. Printout the C source code of the instrumented system. 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.

D. 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 RTI interrupt service routine and at least three other regular functions to profile. You will associate one output pin 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 **`Heap_Allocate`**

```
// MC68HC812A4
unsigned short *Heap_Allocate(void){
unsigned short *pt;
  PORTT |= 0x08;
  pt = FreePt;
  if(pt!=null){
    FreePt = (unsigned short*)*pt;
  }
  PORTT &= ~0x08;
  return(pt);
}
```

```
// 9S12C32
unsigned short *Heap_Allocate(void){
unsigned short *pt;
  PTT |= 0x08;
  pt = FreePt;
  if(pt!=null){
    FreePt = (unsigned short*)*pt;
  }
  PTT &= ~0x08;
  return(pt);
}
```

For RTI 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 **`RTIHan`**

```
// MC68HC812A4
interrupt 7 void RTIHan(void){
unsigned short i; char previous;
  previous = PORTT;
  PORTT = 0x01;
  RTIFLG = 0x80;
  for(i=0; i<=BackData; i++){
    if(LLFifo_Put(i)==0){
      NumLost++;
    }
  }
  BackData++;
  if(BackData==20){
    BackData = 0;   // 0 to 19
  }
  PORTT = previous;
}
```

```
// 9S12C32
interrupt 7 void RTIHan(void){
unsigned short i; char previous;
  previous = PTT;
  PTT = 0x01;
  RTIFLG = 0x80;
  for(i=0; i<=BackData; i++){
    if(LLFifo_Put(i)==0){
      NumLost++;
    }
  }
  BackData++;
  if(BackData==20){
    BackData = 0;   // 0 to 19
  }
  PTT = previous;
}
```

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

Deliverables (exact components of the lab report)

A) Objectives (1/2 page maximum)
B) Hardware Design (none for this lab)
C) Software Design (no software printout in the report)
    none
D) Measurement Data
    Prep part 3) Show the cycle counting of execution speed
    Part A) Show the three results of the execution times
    Part B) Show the execution time measured with the oscilloscope
    Part C) The software profile data, draw arrows on the listings, and data-flow graph
    Part D) The hardware profile data
E) Analysis and Discussion (1 page maximum)

Checkout

You should be able to demonstrate:
Part B. Instrumentation output port.
Part C. Profiling using a software dump.
Part D. Profiling using an output port.

Your software files will be copied onto the TA's zip drive during checkout.

Part C) The programs that implement software profiling
Part D) The programs that implement hardware profiling