Lab 13 Real Time 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
  - Functional debugging (static)
  - Performance debugging (dynamic or real time)
  - Profiling (detection and visualization of program activity)

Review
- Introduction chapter of this laboratory manual,
- Chapter 14 of the M68HC812A4 Technical Summary, in particular look up all the I/O ports associated with SC0: SC0BDH, SC0BDL, SC0CR1...,
- The 6812 Cross compiler documentation, ImageCraft ICC12 manual,
- Valvano Section 2.11 on debugging, Sections 7.5 and 7.6 on the 6812 SCI port
- Read "Developing C Programs using ICC11/ICC12/Hiware" on the TExaS CD or at http://www.ece.utexas.edu/~valvano/embed/toc1.htm

Starter files
- SCI12.H, SCI12a.C, SCI12.C, Rxfifo.H, Rxfifo.C, Txfifo.H, Txfifo.C, Scidemo.C

Background

Debugging is the process of removing errors and verifying the correctness of their software. *Functional debugging* involves techniques that guarantee, given proper input parameters, proper output parameters are generated. It is a static process. Because embedded systems have a tight coupling between its hardware and software the dynamic, or time-dependent, behavior is often critical. *Performance debugging* extends the techniques of functional debugging to include not only what the software calculates, but when it is calculated. In this lab we will study hardware techniques using the scope and LED monitor; software techniques like simulators and memory dumps; and manual tools like breakpoints and print statements. Because of the real time nature of embedded systems, special debugging techniques will be required.

Preparation (do this before lab starts)

1. Breakpoints (functional debugging)

A *breakpoint* can be used to halt execution at strategic places in your program. In this subsection, you will define a conditional breakpoint and employ single-stepping. The purpose of this subsection is to observe the static behavior of the program. In particular, you will observe the sequence of operations that occur when an interrupt is processed. The other objective of this part is to understand the assembly language generated by the compiler. It will be critical to understand assembly language when debugging real time systems.

Download the starter files listed above from the class web site. Modify SCIDEMO.C so that it includes SCI12A.C (interrupt synchronization) rather than SCI12.C (gadfly synchronization.) SCI12A.C will include RXFIFO.C and TXFIFO.C. A crude way to set breakpoints is to define

```
#define bkpt asm(" bgnd");
```

and insert the `bkpt` macro at the places in your source code that you wish to break. The software must be recompiled and reloaded whenever a breakpoint is added or removed. Add the following conditional breakpoint at the beginning of the SCI interrupt. It should break only when processing a receiver interrupt (transmit interrupts will not be halted)

```
if(SC0SR1 & RDRF)   bkpt
```

Compile these programs with the breakpoint inserted. Open the listing file and locate the code that is executed on a SCI receive interrupt (part of SCI12A.C and RXFIFO.C.) Edit the file and print just the portion of the listing file that will be executed during an entire receive interrupt. Circle on the listing the assembly code that implements the line

```
*(tempPt++)=data;  /* try to Put data into fifo */
```

within the function RxPutFifo. Also print out the associated map file. Hand execute this interrupt from the beginning of the interrupt service up to and including the `*(tempPt++)=data;` line within the function RxPutFifo (i.e., show the values in Registers D, X, Y, and SP after each instruction is executed). Also, draw a stack picture at this point. Include the values on the stack as well as the values in Registers D, X, Y, and SP.

2. Global Memory Dump (functional debugging)

Another functional debugging technique is to save strategic information in global memory. Examples of saved information could be input/output parameters of functions, measured data, or the status of important data structures. If the saved information were to include time, then this technique could also be classified as performance

debugging. But, in this part, we will be saving data without reference to the time at which the data was collected. Therefore, this example is classified as functional debugging.

Add debugging instrumentation to TxPutFifo that creates a histogram describing the number of elements in the TxFifo at the beginning of each TxPutFifo call. In particular, while SCIDEMO is running, count the number of times TxPutFifo is called when the TxFifo was empty, had one element, had two elements, had three elements, … etc. Include an entry in the histogram for each possible state of the TxFifo, from empty to full. The x-axis of the histogram will be the number of elements in the TxFifo when TxPutFifo is called, and the y-axis of the histogram will count the number of times this state occurred during the execution of SCIDEMO. This histogram will help us choose the best fifo size. In order to make the debugging minimally intrusive, you can simply create the histogram in global memory and use the BDM debugger to observe it.

3. Monitor using LEDs and an oscilloscope (performance debugging)

Because the 6812 has so many I/O ports, it is likely that some of them will not be required. These unused port pins are available for debugging. An effective tool for real time applications is the monitor. A monitor is an independent output process, somewhat similar to the print statement, but one that executes much faster, and thus is much less intrusive. An LCD display can be an effective monitor for small amounts of information. The Adapt812 has an LED on Port T bit 6. Your software could toggle this LED to let you know the program is running. An LED is an example of a BOOLEAN monitor.

Add debugging instrumentation to the system that provides a visualization of the execution pattern of this system. You will study the dynamic behavior of RxPutFifo and RxGetFifo. The goal is to observe if a SCI interrupt occurs such that a background call to RxPutFifo interrupts a foreground call to RxGetFifo. Interface two LEDs to two output ports (bits 1 and 0.) Increment the port when RxPutFifo is entered and decrement it when RxPutFifo is terminated. In a similar manner, increment the port when RxGetFifo is entered and decrement it when RxGetFifo is terminated. Bit 1 of the port will activate when RxPutFifo interrupts a call to RxGetFifo.

4. Profiling (performance debugging)

Profiling is an example of performance debugging because it measures the dynamic behavior of our software. Profiling is a debugging process that collects the time history of strategic software activities. One of the important software parameters in real time systems is the software response latency. Add debugging instrumentation to measure the software latency to an incoming serial frame. Each incoming frame is 10 bits long. The software latency is defined as the delay from the end of the stop bit to the time the SCI receiver interrupt reads the data from the SC0DRL register. You are free to implement this section however you wish, but you method must allow for repeated measurements.

5. Instrumentation: print statements (no action required)

Print statements are generally used for functional debugging. Using the editor, we could add print statements, called debugging instruments, to software we are debugging. A key to writing good debugging instruments is to provide for a mechanism to reliably and efficiently remove them all when the debugging is done. Consider the following mechanisms as you develop your own unique debugging style:

- Place all instruments in a unique column (e.g., first column), so that the only codes that begin in this column are debugging instruments.
- Define all debugging instruments as functions that all have a specific pattern in their names. For example, make all debugging functions begin with “debug”. In this way, the find/replace mechanism of the editor can be used to find all the calls to the instruments.
- Define the instruments so that they test a run time global flag. When this flag is turned off, the instruments perform no function. Notice that this method leaves a permanent copy of the debugging code in the final system, causing it to suffer a runtime overhead, but the debugging code can be activated dynamically without recompiling. Many commercial software applications utilize this method because it simplifies “on-site” customer support.
- Use conditional compilation (or conditional assembly) to turn on and off the instruments when the software is compiled. When the compiler supports this feature, it can provide both performance and effectiveness.

6. Instrumentation: measurements of execution speed (performance debugging)

In this section you will measure the degree of intrusiveness of print statements. In particular you will measure the average execution time of `OutChar()`. Sometime we can estimate the execution speed of a 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. 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. Unfortunately, cycle counting can not be used in situations where the execution speed depends on external hardware events (like `OutChar()`).

On more complex computers, there are many unpredictable factors that can affect the time it takes to execute single instructions, many of which can not 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 MC68HC812 has a prescaler that can be placed between the E clock (8 MHz) and the TCNT counter. 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 `OutChar()`.

```
void OutChar(char data){
First=TCNT;
   while(!TxPutFifo(data)){};
   SC0CR2=0xAC; /* arm TDRE */
Delay=TCNT-First-12;
}
```

The constant "12" is selected to account for the overhead in the measurement itself. In particular, run the following,

```
First=TCNT;
Delay=TCNT-First-12;
```

and adjust the "12" so that the result calculated in `Delay` is zero. In this technique, we are not measuring the time to call `OutChar()`, but just the internal execution speed.

Write a debugging instrument that measures the execution speed of the SCI function `OutChar()`. Maintain a 32-bit global sum, a counter, and current average. Calculate a new sum, increment the counter, and calculate a new average for each call to `OutChar()`. In order to make the debugging minimally intrusive, you can simply store the results in global memory and use the BDM debugger to observe them. One of the advantages of dumping is that the 6812 BDM module allows you to visualize memory even when the program is running. So this technique will be quite useful in systems with a background debug module (the 68332 has one too.)

7. Instrumentation: dump into array without filtering (performance debugging)

One of the difficulties with print statements is that they can significantly slow down the execution speed in real time systems. Many times the bandwidth of the print functions can not keep pace with the existing system. For example, our system may wish to call a function 1000 times a second (or every 1 ms). If we add print statements to this function that require 50 ms to perform, the presence of the print statements will significantly affect the system operation. In this situation, the print statements would be considered extremely intrusive. Another problem with print statements occurs when the system is using the same output hardware for its normal operation, as is required to perform the print function. In this situation, debugger output and normal system output are intertwined.

To solve both these situations, we can add a debugger instrument that dumps strategic information into an array at run time. Write a debugger instrument that saves the ASCII character and the time (TCNT value) at each call to TxPutFifo. Because we care saving both the data and the time the data was generated this is an example of a real time profile. Again, you can simply create the dump in global memory and use the BDM debugger to observe it. Allocate enough space to collect the character and TCNT value for the first 10 calls to TxPutFifo.

8. Simulation (performance debugging) (optional +5 point bonus)

A simulator is a software application that simulates or mimics the operation of a processor or computer system. Unlike TExaS, most other simulators recreate only simple I/O ports and often do not effectively duplicate the real time interactions of the software/hardware interface. On the other hand, they do provide a simple and interactive mechanism to test software. Simulators are especially useful when learning a new language, because they provide more control and access to the simulated machine than one normally has with real hardware.

Many of the software systems in this course can be run on the TExaS simulator. You should view the two movies located on the TExaS CD. The first, called ReadMe.exe, contains general information about developing assembly language software using TExaS. The second, called ReadMe2.exe, contains specific information about developing ICC12 C programs using TExaS. As part of this optional preparation, run the TUT4 tutorial located in the ICC12 folder. It is very similar to the Scidemo.C example that you are debugging in this laboratory.

9. Instrumentation: dump into array with filtering (functional debugging) (no action required)

One problem with dumps is that they can generate a tremendous amount of information. If you suspect a certain situation is causing the error, you can add a filter to the instrument. A filter is a software/hardware condition that must be true in order to place data into the array.

10. Thread Profile (no action required)

When more than one program (multiple threads) is running, you could use this technique to visualize the thread that is currently active (the one running). For each thread, we assign an output pin. The debugging

instrument would set the corresponding bit high when the thread starts and clear the bit when the thread stops. We would then connect the output pins to a multiple channel scope to visualize in real time the thread that is currently running.

Procedure (do this during lab)

1. Breakpoints (functional debugging)

In order for breakpoints to be active, the software must be started from the Kevin Ross BDM debugger using the commands

```
reset
load scidemo.s19
g f000
```

Run the system with the breakpoint inserted. After a breakpoint, registers and memory can be viewed or changed. To single step you can execute t. Use this single step function to verify the results you calculated as part of the preparation. To restart execution after the breakpoint, type g. To remove a breakpoint, you will have to edit the source code, compile, and load again.

2. Global Memory Dump (functional debugging)

Run this functional debugging technique for three sizes of the TxFifo. Choose one fifo size that is very small, so that it is often full, and choose one fifo size that is large enough so the fifo is seldom empty. Observe the map file, and be careful not to overflow the available 1024 byte RAM space. The compiler will not generate an error if you try to define more than 1024 bytes of RAM information. Discuss in your write-up what fifo size is most appropriate for this application.

3. Monitor using LEDs and an oscilloscope (performance debugging)

Run the debugging system with the two LED monitors. Observe the dynamic behavior on a scope. Does the SCI interrupt call to RxPutFifo reenters a foreground thread call to RxGetFifo? Discuss in your write-up why the RxGetFifo function disabled interrupts.

4. Profiling (performance debugging)

Run the profiling instruments to measure the software latency to an incoming serial frame. Perform repeated measurements and calculate the average and maximum latency.

6. Instrumentation: measurements of execution speed (performance debugging)

Measure average execution speeds of `OutChar()` for three sizes of the TxFifo (the same three values used in part 2.) Also measure the average execution speed of `OutChar()` using the gadfly synchronization method (using the file SCI12.C.)

7. Instrumentation: dump into array without filtering (performance debugging)

Collect data on the character and time for the first 10 calls to TxPutFifo.

8. Simulation (performance debugging) (optional +5 point bonus)

Run the exact object code for part 7) on the TExaS simulator. You can run TExaS and open the TUT4.UC and TUT4.IO files located in the ICC12 folder. Click on the TUT4.IO file and activate the CRT command in the IO menu. Using this dialog, change the CRT baud rate from 500000 to 38400 bits/sec. Next activate the LoadS19… command in the Action menu. Find and open the S19 file from your part 7). The TExaS simulator should be ready to go. You can use the ViewBox to observe global memory. Just like part 7), collect data on the character and time for the first 10 calls to TxPutFifo. Discuss the similarities and differences between using real hardware and simulation.

Checkout

You should be able to demonstrate:

2. Collection of the histogram.
3. Determination of whether the RxPutFifo reenters a call to RxGetFifo.
4. Measurement of software latency.
6. Measurement of execution speed.
7. Real time profile.
8. Simulation of real time profile (optional extra credit).

Hints

1) One possible method that could be used to measure software latency involves the use of input capture to measure the beginning of the stop bit, and the TCNT timer to measure when the SC0DRL is read.
2) Another possible method that could be used to measure software latency involves the use of one channel of the scope to measure the end of the stop bit, and an output port pulse (and another scope channel) to measure when the SC0DRL is read.
3) If an assignment is unclear, ask your TA for a clarification.
4) To use breakpoints on the 812A4, we need to be running in Special Mode.

- insert "bgnd" instruction at the place you wish to break
- compile to executable
- download to EEPROM
- start the program using g F000 (do not hit the reset button on the Adapt812)
  when the break occurs, there will be no clue that it occurred, except the "r" command will now work. FYI the reset button places the board in Normal Mode. Starting the 6812 from the Kevein Ross BDM "g" command places the 6812 in Special Mode