EE247 Tutorial: Interrupts and Multitasking: The Problem

Background:

This document has been drafted as an exercise for students in EE247 Programming for Embedded Systems, currently being offered for the Fall 2020 semester at Wilkes University. The course uses the NXP MC9HCS08SH8 microcontroller and the available 'SH8 "Demo" boards for student exercises to support the course. (The "hcc_terminal" from NPX/Freescale originally distributed for the JM60 family has been utilized in this course as an interactive environment.)

So far in the course we have used the "terminal" in a polling mode. It was demonstrated earlier in the semester that a long-running process in the main loop, such as a busy-wait timing sequence to blink an LED, can seriously disrupt communications since the polled terminal process can't process characters quickly enough. It was shown that using an interrupt for the serial port receive, and having that interrupt call the terminal process, the problem could be mitigated.

After starting to utilize interrupts for the serial port receive as well as timers, it might seem that problems with timing are now a thing of the past. They are not. The purpose of this exercise is to demonstrate that. Subsequent developments and discussion will explore ways to get around the problems.

The problem:

With the serial port (SCI) receive interrupt used to process incoming characters, activity in the main loop will no longer interfere with the execution of "commands" that the user might issue. However, suppose those commands happened to take a while? Here are some examples of how that could happen:

1. The A/D converter might be sampled. Right now we use a busy-wait loop to wait for the "conversion complete" (COCO) flag before fetching the value and continuing computation.
2. The command may issue a "print" call. Print calls the serial port version of the "put character" (putchar) function. That function is "blocking" in the existing serial port interface files. So, there is a busy-wait until the transmit data register is empty (TDRE) flag goes on to allow the next character to be transmitted. At the 115200 bps rate we have been using, that is a delay of only 87 uS per character, not terribly long. But suppose the print string is lengthy? Or suppose the more common speed of 9600 bps is used? A 100 character message would take 104 ms. That's a long time. Or if slower, even longer.
3. The command might invoke a computational procedure that takes a while, such as Digital Signal Processing (DSP) algorithms, especially if floating point is used. This is only an 8 bit processor. This kind of math is going to be slow.

So, our purpose here is to demonstrate this problem. The starting point will be the "terminal" program with which we started the course. You should have already had this running. If you need help getting going, see the document "Initial Lab Exercise: Terminal Program." (It is at < http://www.jbgilmer.com/EE342/EE342Terminal%20SH8.pdf >. I'll put it on the EE247 page.

The Exercise:

The following is a step by step description of the exercise. The first few steps duplicate what we already did in class earlier to motivate our need for and use of interrupts.

1. **Illustrate the Original Problem:** This step duplicates what was done in an in-class exercise earlier in the semester showing the need for interrupts. We will add a lengthy activity in the main loop which will consume most of the microcontroller's processing resources. The main loop should currently look like this:

```
for(;;) {
    c=SCIS1;
    terminal_process(); /* services terminal, dispatches commands*/
    DisableInterrupts;
    __RESET_WATCHDOG(); /* feeds the dog */
    EnableInterrupts;
} /* loop forever */
```

Add a long integer for a loop counter to main's local variables (after the char). We use a long to make the counting slow. Then add code to blink the LED at PTBD6 slowly on and off. First, modify the initialization of Port B to make both pins 6 and 7 outputs:

```
char c;
long i;
```

(and later in the code:)

```
PTBDD=0xc0;  /* rather than 0x80 */
```

Modify the main loop to include "blink the LED". We feed the dog inside the timing loops to both slow down the delay and make sure the dog doesn't go off.

```
for(;;) {
    c=SCIS1;
    terminal_process(); /* services terminal, dispatches commands*/
    PTBD_PTBD6=1;
    for(i=0;i<10000;i++)__RESET_WATCHDOG();
    PTBD_PTBD6=0;
    for(i=0;i<10000;i++)__RESET_WATCHDOG();
    DisableInterrupts;
    __RESET_WATCHDOG(); /* feeds the dog */
    EnableInterrupts;
} /* loop forever */
```

Compile, "debug" (download your code into the microcontroller and run) with PuTTY running and the serial port set for 115200 bps. The LED on PTB6 will be blinking at about 1Hz. (If you want it slower, make the constant in the counter loops bigger than 10,000.) Try typing in commands, either "help" or "led". If you do it slowly, it works. If you speed up your typing, the terminal will miss characters. That's because the terminal process is only called very occasionally, and you type faster than it can be called with that slow LED regulating the cycle speed of the main loop. This is "The Original Problem." The next step is the fix for this problem.

2. **Fix the Original Problem:** The fix for the problem above is to add an interrupt for the Receive Data Register Full (RDRF) of the serial port. That's what we will do here, leaving in place the slow timing loop LED blinking in the main loop. First, modify the function `TERMIO_Init_16M_115200` in the file "SCI_Functions.c" to enable the RTIF interrupt. Modify the following statement as shown:

   ```
   SCIC2  = 0x2C;  /*turns on transmitter, receiver, was 0x0c*/
   ```

   Next, delete the call to terminal_process in the main loop and put it in an interrupt service routine for the RTIF interrupt. (The two statements in the ISR are deleted, or commented out, in the main loop.)

   ```
   interrupt 17 void rtdf_isr(void){
       c=SCIS1;
       terminal_process(); } /* services terminal, dispatches commands*/
   ```

   Now then the code is compiled and the executable image downloaded and run, there is no problem. Or seems to be no problem. When a character comes in, the RDRF flag goes up and the ISR for it interrupts the main loop just long enough to handle the command input (and anything attendant to it). Yes, the main loop timing is affected ever so slightly. Not enough to matter, right? Well, maybe. We'll see. This at least fixed the "Original problem", that we were losing keystrokes because the main loop was not allowing us to poll the terminal_process often enough. (We could have put a call to terminal_process inside the timing loops; that would also have fixed it, but we are trying to illustrate the general issues.)

3. **Illustrate the Remaining Problem:** Let us suppose that the "process" in the main loop, currently blinking the LED, is important. Maybe something like "Keep the wings on the airplane." To do that we'll make a couple of modifications to what we have. First, we'll speed up the blinking to represent a process that needs to go relatively fast, say, by a factor of 10. So, reduce the timing loop limits from 10,000 to 1,000. You will still be able to see the LED blink. But you will find it much easier to notice if something disturbs the blink pattern.

   ```
       PTBD_PTBD6=1;
       for(i=0;i<1000;i++)__RESET_WATCHDOG();
       PTBD_PTBD6=0;
       for(i=0;i<1000;i++)__RESET_WATCHDOG();
   ```

   Now, we will change the serial port to use the more common 9600 bps speed. Create a new SCI initialization function in SCI_Functions.c of "`TERMIO_Init_16M_9600`". You can copy and paste the 115200 bps function. Change SCIBDL setting to 104 (as below).

   ```
   void TERMIO_Init_16M_9600(void) {
     SCIBDH = 0;
     SCIBDL = 104;
     SCIC1  = 0;
     SCIC2  = 0x0C;  /*turns on transmitter, receiver*/
   }//end TERMIO_Init_16M_9600
   ```

After doing this, you will need to restart PuTTY at 9600 bps. You should be able to do the same things as before, but now your serial port is operating at the slower (and more common) speed. (I found that I got garbage characters when I made this change. I had to fiddle with the clock adjustment in the function "`ICS_FEI_16M`" to get it to work. Part of that is because the clock had to be adjusted originally to work at 115200 bps. The low resolution of the SCIBR register at such high speeds meant that the clock trim had to be fixed. Now, it needed to be fixed back from 0x96 to 0xA6 to work correctly at 9600 bps.)

Now, the payoff: Type in the command "help" but do not hit return yet. Position your finger over the return key. Watch the blinking LED on your demo board. Now hit "return" while watching the LED. You should notice that it misses its rhythm. It noticeably turns off for a fraction of a second, or it turns fully on for a fraction of a second. It has been interrupted. If you were running the serial port at 300 bps, the effect would be more noticeable.

Also, watch the screen as you execute the "help" command. It's not quite instantaneous. I figure 155 characters at 9600 bps, 10 bits per character (including start and stop bits) means that printing help takes 160ms. That's 160 ms during which the LED can't be doing its job of holding the airplane wings on. That's a problem.

Discussion:

You may be thinking, "Blink the LED inside an ISR." Let's suppose we make it the RTC ISR. We need to make the period of the ISR long enough that we can execute all of those loops. (The timing loops which really are just wasting time represent important processing that the machine needs to do.) Now, youy might think the ISR will go off as scheduled. But, it won't! While the RTDF interrupt is going, the RTC ISR cannot interrupt! We will have the same kind of effect. And, while the RTC ISR is running, the RTIF interrupt can't fire, and we will miss characters. Now we have BOTH problems, in an unpredicatable manner! (If we were holding this demo in the classroom (ahem – laboratory) we might try some of these things.)

Our basic problem is that both the "Blink the LED" process in its entirety, and the process of printing the "help" response at 9600 bps, take too long! Ideally an ISR comes in, does its business quickly, and gets out. It doesn't tie up the processor doing busy-wait stuff. In this case, the busy-wait is waiting for the serial port. In other cases there might be a lengthy busy-wait waiting for the ADC. Or other things.

The bottom line is that anything lengthy should not be in an ISR. For us, that means printing through the serial port. Processes that take a long time need to be somewhere else: the main loop. In the main loop they can be interrupted. As long as an interrupt is brief, that will be OK. It is long interrupts that are the problem. So, what we need to do to fix the problem we have now is to make the process of printing interrupt based.

On more sophisticated processors, there are "interrupt priority" systems. Each interrupt has a priority level. Typically there are 7 levels, with level 0 being "the main loop" (no interrupt) and level 7 being the highest level interrupt. The idea is that the most important interrupts can interrupt less important ISR's. The highest priority interrupts should be very fast. The lower priority interrupts could be slower, since they can be interrupted.

For the MC9HCS families of processors, there is only one level of interrupt. When an ISR is called, the interrupt flag in the status register (inside the CPU) is set to 1. This blocks additional interrupts until the flag is cleared (enabled). Coming out of reset the flag is set so that the program can check the reset reasons before resuming normal business, and the main program can do some things with no fear of interruption until it is ready to go. Then main() can allow interrupts to commence. There are priorities associated with the different interrupts, but they only affect which interrupt gets serviced first if more than one is pending. An interrupt can't interrupt another ISR. (Yes, you could clear the interrupt flag in the ISR to allow another interrupt, but we have so little stack RAM that we'd be asking for serious trouble.)

What's up Next:

We will do something about the problem, but first a discussion of tasks and processes.