EE247 Flags and Semaphores

March 30, 2020

Background:

We have seen in earlier projects and tutorials how useful interrupts can be. Because the computer we are using, the MC9S08SH8, has only one interrupt level, we generally want to put anything that takes a while at the "zero" interrupt level, that is, in the "main program" that runs while not in an interrupt service routine. We don't want interrupt service routines to take much time at all, because while they are running, not only is the main program blocked but all other interrupt service routines as well.

So, the strategy generally used is to avoid, in interrupt service routines, anything that requires waiting or a lot of time. That means partitioning tasks do things, such as the tasks our command callback functions do, into separate pieces. The command callback function, called out of a receive data on the serial port interrupt, are in effect part of the interrupt service routine. Yes, we created a non-blocking print, but it's very memory intensive; we may not be able to afford that. Suppose we needed to use an ordinary, blocking, print? Then, what we would do is partition the command function into two parts: The command function proper, which would avoid printing, and the remainder, which would do anything that requires waiting. The problem is, how does the command callback function signal to the function that does the rest of the work what it is supposed to do? That's where flags come in.

This document looks at that issue by building an example command function for the A to D converter. For that, we will need to partition three ways, as we'll see.

The original function and issues:

Here is the original function code for a simple "adc" command that reads the pot.:

```
/* function to report ADC results */
static void cmd_adc(char *param){
  int i;
  char s[6];
  ADCCFG=0x40; /*10 bit mode*/
  param++
  /* collect pot value */
  ADCSC1=0;
  while(ADCSC1_COCO!=1);
  i=ADCR;
  itoa(i,&s,5);
  print("Pot=");
  print(s);
  print("\n\r");
}
```

There are three distinct steps. First, the ADC conversion is initiated. Then we wait. Then the result becomes available. The number is converted into a string. Then we do three different calls to "print" to return the value to the user. That involves waiting for the various characters to go out (assuming we are using a non-buffered, blocking, print).

Breaking it up; using a Flag:

So, we will separate each of these. Separating the first step from the second is easy. The ADC has its own interrupt, that can be set to trigger when the "Conversion Complete" flag (ADC_COCO) goes to one. When we store the channel number into ADCSC1, we just need to also enable ADCSC1_AIEN, the interrupt enable flag. That's all the command callback function needs to do! It becomes simply:

```
/* function to report ADC results */
static void cmd_adc(char *param){
  int i;
  char s[6];
  param++;
  ADCCFG=0x40; /*10 bit mode*/
  /* start collecting pot value */
  ADCSC1=0|0x40; /* channel or'ed with interrupt set */
}
```

If we are going to enable that interrupt flag, we need an ISR to catch it. That's where we pick up the data. But, inside an ISR we don't want to print! So, what to do with it? We'll put it somewhere. Somewhere that can be accessed from the main function (inside the main loop). That means a global variable.

```
int adcResult;
interrupt 23 void adc_isr(void){
    adcResult=ADCR;  /* this also clears COCO flag */
}
```

Then, we need something to do the print. Something in the main loop that will run when no ISR is active:

```
  /* main loop */
  for(;;) {
    char s[6];  /* or earlier */
    if( /* ready to print ADC stuff */){
        itoa(adcResult,&s,5);
        print("Pot=");
        print(s);
        print("\n\r");
    }
    DisableInterrupts;
    __RESET_WATCHDOG(); /* feeds the dog */
    EnableInterrupts;
  } /* loop forever */
```

The problem is, how do we know when the ADC results are to be picked up and printed? We need a flag for that. Something to test whether we should do the print. If we don't have a test, printing the value occurs continuously. We don't want that. Here are two strategies:

1) Only print if the value changes. To do that we would need a variable accessible in main to the previous value of the output, and print if it changes, and by the way, modify the referenced last value. Sort of like we would do for the pushbuttons. The problem is that if we check the A/D and get the same number, we get no print at all!
2) Use a "flag" to tell the main loop that there is something to print. This is better.

So, using that method, the ISR would set the flag when it picks up the data:

```
int adcResult;
char adcFlag=0;
interrupt 23 void adc_isr(void){
    adcResult=ADCR;  /* this also clears COCO flag */
    adcFlag=1;
}
```

And the main loop would check (and clear the flag when it prints):

```
  /* main loop */
  for(;;) {
    char s[6];  /* or earlier */
    if(adcFlag){
        itoa(adcResult,s,5); /* s is a pointer to the chars. */
        adcFlag=0;
        print("Pot=");
        print(s);
        print("\n\r");
    }
    DisableInterrupts;
    __RESET_WATCHDOG(); /* feeds the dog */
    EnableInterrupts;
  } /* loop forever */
```

It's a good idea to clear the flag as close after the picking up of the data as possible. It is possible for the ISR to go off before you clear the flag. A more sophisticated program might have a buffer of ADC results, and could hold a few of them ready to print on successive passes. That wouldn't really apply here. In clearing the flag, the main program is saying it is committed to printing the value.

A more complicated case: A semaphore

We have been discussing printing earlier, and problems related to printing. Recall that if we had a buffered print, and were assured we would never overfill the buffer, we could call that print out of an ISR (or a function like a command function called out of an ISR) and not have

large delays. But there may be times, like the print for "help," that we can't use that buffered print because the buffer will (or even just might) overflow. One way to solve that would be to have two print functions, one for "help" (or printing an image to the screen, like the dog chasing the stick), and the other for quick prints out of an ISR (or command function).

The problem is, you would like to avoid having both print functions going at the same time! Characters could end up interleaved. That just makes a message unreadable. So, you need to "guard" access to the character sending resources so that only one version of "print" can go at a time. The tool used to do that is a flag. But, it is a special kind of flag called a "Semaphore". This harkens back to the old railroad days when such signals, with a horizontal arm to mean "stop" and a vertical arm position to mean "go", were used to guard railroad intersections from simultaneous use by two trains. (Railroad trains are made up of, among other things, baryons. Those particles, unlike photons, must obey the Pauli Exclusion principle. In effect, two trains cannot occupy the same place at the same time. If they attempt to do so, bad things happen.) The "semaphores" were intended (if obeyed) keep the bad things from happening.

That's very similar to what we want to do. In our case, the two semaphores (or more, one per track approaching) are merged into one variable. There are two print functions. The semaphore needs to indicate who (or nobody) has permission to send out characters. Let us say that flagPrint is the name of the semaphore, and that "0" indicates that nobody is printing, "1" indicates that the (blocking) old version of print is printing, and a "2" if the new buffered version of print is printing. Initially we would come out of reset with the semaphore set to "0". Each way of printing needs to check the semaphore before shoving characters into the serial port. The old print is the easiest. We check on entry, and if it is "2" we simply wait. Once in, we set the semaphore to "1", print, then set it to "0" when we are done:

```
void oldprint(char *s)
{
char c; /*added*/
  while(flagPrint==2);
  flagPrint=1;
  while(*s)
  {
    /*while(*s != (char)putch(*s))*/
    /* Replace with just a CALL TO PUTCH */
    c=(char)putch(*s);
    s++;
  }
  flagPrint=0;
}
```

Doing this for the buffered print is trickier! It is actually in the TDRE ISR that characters are sent out! We need to turn this off if we go to start printing and the old print that is blocking is in the middle of a print job! We are not allowed to send characters yet! (We don't want to change the actual print function because we don't want to block it. We will still accept characters into the buffer.)

```
interrupt 18 void tdre_isr(void){
    if (flagPrint==1){ /* if print busy exit */
       SCIC2_TIE=0;
       return;}
    char c=SCIS1;
    SCID=bufFer[bufStart++];
    if(bufStart>=192)bufStart=0;
    if(bufStart==bufEnd){  /* if done printing */
        SCIC2_TIE=0;
        flagPrint=0;}
 }
```

So, what we do is we turn off the ISR. That prevents it from interfering with the other print job. But, how do we turn it back on? If the ISR is turned off, it can't be checking to see if it is time to turn on again. Someone else has to check and see if the coast is clear. Meanwhile, characters accumulate in the buffer, waiting.

The easiest way to do this is have the main loop check and see if the TDRE can be turned on:

```
  /* main loop */
  for(;;) {
    char s[6];  /* or earlier */
    /* Check to see if can turn printing back on */
    DisableInterrupts;
    if(bufStart!=bufEnd&& flagPrint==0){
        flagPrint=2;
        SCIC2_TIE=1;}
    EnableInterrupts;
    if(adcFlag){
        itoa(adcResult,&s,5);
        adcFlag=0;
        print("Pot=");
        print(s);
        print("\n\r");
    }
    DisableInterrupts;
    __RESET_WATCHDOG(); /* feeds the dog */
    EnableInterrupts;
  } /* loop forever */
```

More sophisticated methods would be to do this at the end of the old print function – it could check to see if the buffer is empty and then turn the ISR on and set flagprint to 2 instead of just changing flagPrint to 0.

Notice we want to preclude interrupts while we are flipping the semaphore! An interrupt between checking and setting can be a big problem! As shown, the old print is vulnerable.

A second unrelated topic: Zero Page

If we make the print buffer as large as we might like, we use up lots of RAM. One way to help ourselves to more RAM is to start using zero page. This has the added benefit of making RAM references smaller and quicker. (Instructions can use the "direct" mode instead of "extended mode" in reading a variable. So, an LDA x is "B6 80" if x is in zero page at 0x80, and "C6 01 00" if x is in the main global memory at 0x0100.)

The memory map in the linker file "Project.prm" identifies different parts (segments0 of RAM. There's a "default" segment and a "zero page" segment.

A statement **"#PRAGMA DATASEG MY_ZEROPAGE"** says to the compiler, that all variables (globals or static locals) from here on are to be put in zero page. To resume putting variables in the default segment, use **"#PRAGMA DATASEG DEFAULT_RAM"**. (A Pragme is an instruction to the compiler.)

A final caveat:

The Zero Page stuff above has been tested in the buffered print program, but the flag and semaphore code, as you see it in this document, has not. I've done similar things before, but not here. There just has not been time. The point of this document is to discuss the concept, not as a recipe. I'll get to it when I find time to do so, but that has not happened yet.

Also, there are lots of issues that have not been addressed. Notice that the semaphore blocks the buffered print buffer from emptying, possible for quite a while. Could that be a problem? Well, yes, because the buffer could overflow if enough stuff is printed to it. Could that happen? Well, yes, if ISR and ISR called functions do printing. The solution to that is to handle all printing like the printing for the ADC command was: the message to be printed (and the information calculated to go into the message) don't get printed except in the main loop. And ISR's don't print at all. And use a "buffered" print only for absolute emergency messages perhaps.

These are all issues that can come up in the context of real time embedded systems. Our course just can't go but so deep. (Things get even more complicated if you have multiple processors, especially if they share access to the same memory. That gets messy. And much more advanced than we can address here.)