EE247: Buffered Printing

March 24, 2020

Background:

This is a laboratory exercise intended to follow-up previous material in the documents "EE247 Tutorial: Interrupts and Multitasking: The Problem" and "EE247: On Multitasking." The first of those documents develops a demonstration of the problem seen when printing (slowly) while trying to do other processing. The basic problem is that "print" is called out of an Interrupt Service Routine, and is a "blocking" version of print. So, nothing else can happen on the computer while a print is underway. The second of those two documents discusses the problem further, and outlines some of the issues attendant to "multi-tasking." This document assumes familiarity with those two preceding. This document develops one solution to the problem: a buffered print. The print function is no longer blocking. Calling it dumps characters into a buffer, and those characters are sent out by another process that empties the buffer. It is expected that students will follow this exercise step by step with their microcontrollers (MC9S08SH8 on its demo board, supported by the CodeWarrior for Microcontrollers Integrated Development Environment (IDE).

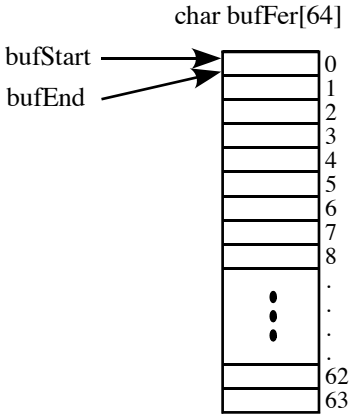
A Circular Buffer:

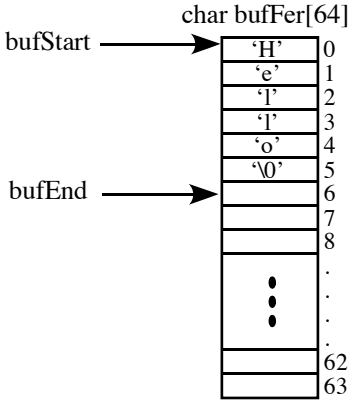
The buffer we want needs to act like a "First In First Out" (FIFO) memory.

The problem is how to implement the buffer. There are three elements. One is the buffer itself. Then we need a way to load it. We'll call that the "print" function. (That is, the new, non-blocking version of the print function.) The second is the function that empties the buffer by sending out the characters, which will turn out to be an Interrupt Service Routine (ISR).

We'll start with the buffer itself. What is it? Whatever it is, it has to hold a bunch of characters. It also has to be made out of RAM, since that's what our computer memory actually is. (We can't use ROM; anything written to ROM falls on the floor.) So, we'll set aside a "block" of memory. It is convenient to use sizes that are a power of 2, so let's make it 64 bytes.

At any given time the number of bytes (characters) can vary. We can use two indexes, called bufStart and bufEnd, to keep track of where the current contents of the buffer are. A diagram is shown at right. Initially, the buffer would be empty, with both bufStart and bufEnd both zero, pointing to the first (and next available) empty space in the buffer. Suppose the main process then prints the string "Hello!" What the print function would do is put the contents of that string, one byte at a time, into the buffer. At the end of doing so, it looks as shown at right.

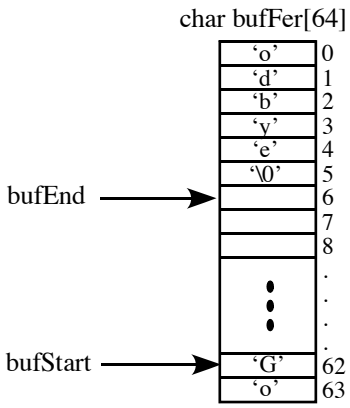




What would happen next is the process to empty the buffer would start up, and the first few characters would be sent out. The bufStart index would start to move such that it begins to catch up to bufEnd. If nothing is printed, the bufStart pointer eventually catches up with bufEnd. When the two are equal, the print buffer is empty.

But let's suppose something else got loaded into the buffer before that happens. Say, "sw1=0". Then, 's' would follow the '\0' in the buffer element pointed to by bufEnd, and bufEnd would move down one for each character as they are loaded.

What makes this a "circular" buffer is that, when bufEnd reaches 64, just beyond the last word of the buffer (at 63) it is reset to 0. So, it never really comes to the end of the buffer. The buffer is a circle of bytes 64 characters long. The same thing is done with bufStart. So, bufStart continues to chase bufEnd as long as there are characters in the buffer. The figure at right shows the situation where bufEnd has rolled around to the top but bufStart has not.



Data Structures – The Buffer:

So, to build the buffer, what we need are a 64 byte (character) buffer, and two "indexes". These all need to be globals, since they have to be accessible from "print" or the ISR that empties the buffer, and the contents need to be persistent.

```
/* Globals to support buffered print */
char bufStart=0,
char bufEnd=0;
char bufFer[64];
```

But, where should the declaration / definition of these variables be? We are going to need to modify the "print" function which is currently in hcc_terminal.c. If we are replacing it with a print function that will use the same name, it makes sense to modify the file hcc_terminal.c. (Be sure to archive the existing file for future reference when we are done with this exercise!) So, we will put these new globals into hcc_terminal.c after the other globals there. We will also comment out the existing print function and replace it with out new one.

The Print Function:

Now we need a function to print that, instead of callinbg getchar, will simply load up the buffer starting at the current index value of bufEnd, and advancing forward (in a circular manner) through the buffer. We'll restructure the existing print function to do so.

```
void print(char *s){
  while(*s)bufFer[(bufEnd++)&63]=*(s++); /* put string into the buffer */
  SCIC2_TIE=1; }                         /* enable TDRE interrupt */
```

Notice the use of "++" within the statement. This is called a "side effect." The statement not only modifies the buffer, it also modifies s and bufEnd, adding one to each after the bufFer is written. Tricky! When people are using side effects, especially in a statement that does looping (as here) if can be quite difficult to figure out what is going on. The &63 sets 64 back to 0. In fact, this is so clever it doesn't actually work! Hard to figure out why!

OK, let's rewrite it to be a little less opaque:

```
void print(char *s){
  while(*s != 0){
     bufFer[bufEnd]=*s;              /* put string into the buffer */
     s++;                           /* increment string pointer */
     bufEnd++;                      /* increment buffer end pointer */
     if(bufEnd>=64)bufEnd=0;        /* make buffer circular */
   }
 SCIC2_TIE=1;                       /* enable TDRE interrupt *
}
```

Emptying the Buffer:

You will noticed that we turned on the interrupt flag just before the print function exited. That's going to immediately trigger the ISR (unless it was already busy sending out characters when print was called). The ISR needs to send characters until the print buffer is empty. But, it only has to send one as ta time. it does not have to loop. It's important to read the details about the TDRE flag in the manual. How do we turn it off? After the ISR gets called, we need to turn that flag off. It will set itself again afterwards when the transmit buffer is again empty, but that will cause a different invocation of the ISR. Each ISR call only needs to handle one character. Doing that is fairly straightforward:

```
interrupt 18 void tdre_isr(void){
    char c=SCIS1; /* needed to clear TDRE. NOT SCIS1_TDRE, but whole reg! */
    SCID=buffer[bufStart++]; /* send data and clear TDRE */
     if(bufStart>=64)bufStart=0;
     if(bufStart==bufEnd)SCIC2_TIE=0; /* turn off int if empty */
}
```

Both the print function and the ISR really should not be in hcc_terminal because hcc_terminal does not have an include for the MC9S08SH8 registers. It is meant to be "generic", not caring what processor is being used. Indeed, hcc_terminal actually comes from a suite of files used with demo programs for the MC9S09JM60, a version of this same family that had a USB port, with which hcc_terminal was used. To be able to compile both of these functions we need (earlier):

```
#include "MC9S08SH9.h"  /* needed to use SCIC2_TIE, SCID */
```

Run it!

With this, it ought to work. Compile and debug (download) and start it running with PuTTY up. You should see the welcome message appear. "led" should work. "adc 1" should return something to the screen for the value of the pot. Hurray! It works. Now try the "help" command. Uh oh. Mine just printed " 1 for pot / 2 for photo". What happened? You could try setting breakpoints and watch print and the isr running, but then you might not see what happened because debugging one step at a time gives the serial port plenty of time to keep up with print. At full speed, the isr can't keep up. The print buffer overflows. More precisely, bufStart overtakes and passes bufEnd! 64 characters, most of the "help" message, is lost.

So, what do we do? Make the buffer bigger! I cranked it up to 128. But, the same thing happens! It seems I had been overrunning the buffer twice! We could go up to 256, but we need to make bufStart and bufEnd unsigned integers instead of chars if we do that. Now I get a different error: I run out of RAM! Finally, with a buffer of 192 characters, it worked. Try it!