ECE 341 - Lab #7

Collin Heist

March 28, 2019

Contents

Listings

1 Introduction

The purpose of this lab is to learn about asynchronous communications and how to use them to communicate with a microcontroller. We'll be using *UART* to communicate in real time between a computer terminal and the microcontroller. We'll be using a *library* provided to us that has quite a few helpful functions for writing to and reading from the UART interface.

The actual task we'll be performing in this lab is very similar to **Lab #5** and **Lab #6** combined. We'll be using entirely interrupt driven code to operate the motor. This includes a one millisecond timer interrupt for precise control of the motor's movement, and a change notice interrupt for detecting whether or not the buttons are pressed (and their state needs to be parsed). We'll combine this code with last week's lab, where we interacted with the LCD. In this lab, we'll be writing the current state of the motor, as dictated by the state of the buttons, to the LCD. But, we'll also be parsing input from the UART interface on the computer, and any inputs provided by the user as to the motor's state will also be written to the LCD. This makes the LCD a *shared resource*, and thus we'll need to account for possible interruptions to writing to this resource, and take steps to avoid problems involved.

In order to accomplish this, we'll be using a handful of peripherals. Of course the on-board interrupt controller in order to use the one millisecond timer, as well as the change-notice detection on the buttons. We'll be using the **PMP** from last week to control the LCD. And the only new peripheral we'll be using is the UART controller. This peripheral will need to be initialized with a few parameters, like the baud rate (transmission frequency), and the parity. These will need to be the same on both the microcontroller side, and the PC side of the transmission. This is to ensure that both sides can send and receive successfully.

2 Implementation

The amount of code needed for this lab was quite small. To begin, I changed the **system_init()** function to initialize the UART peripheral for our desired speed. That change is shown here:

Listing 1: System Initialization Function

```
void system_init() {
    Cerebot_mx7cK_setup();
    initialize_LCD();
    initialize_uart1(19200, ODD_PARITY);

    // Rest of system_init
    ...
```

This section of code is fairly simple, as all it does is open turn on UART1 = for our desired baud rate and parity, both of which were given to us in the lab handout, and it also initializes the LCD. The rest of the **system_init()** function is the same as **Lab #5**, where the timer 1 interrupt it turned on and set to one millisecond, and the change notice interrupt is set to detect button 1 and 2.

The next change that needed to be made was to the **decode_buttons()** function. For this lab, when the buttons are pressed, the corresponding 'state' of the motor (direction, speed, mode) needs to be displayed on the LCD and sent to the PC via UART. So, the following changes were made to this function:

Listing 2: Changed Button Decoding

```
void decode_buttons(unsigned int portG) {
    unsigned int btns = portG >> 6;

    step_mode = btns & 0x01;
    dir = (btns & 0x02) >> 1;
    RPM = (btns & BIT_0) ?
        (btns & BIT_1 ? 10 : 25) : 15;

    char buffer[15];

    if (dir == CW) { strcpy(buffer, "CW "); }
    else { strcpy(buffer, "CCW "); }

    if (step_mode == HS) { strcat(buffer, "HS "); }
    else { strcat(buffer, "FS "); }

    if (RPM == 10) { strcat(buffer, "10"); }
    else if (RPM == 15) { strcat(buffer, "15"); }
    else { strcat(buffer, "25"); }

    reset_clear_LCD();
    put_string_LCD(buffer);
    putsU1(buffer);
}
```

The first few lines are unchanged. The value of the buttons are read from **PortG**, and those are used to determine the corresponding step mode, direction, and RPM. The actual correspondence of these is provided by the **Lab #5** handout. The changes begin with the character array. Here, one large character buffer will be used to *construct* the message that is sent to the LCD and the UART.

After declaring the character array, I begin constructing the string. Since the format of the string is direction, then mode, then speed, I begin by looking at the value of the global variable **dir**, and depending on the value inside, I copy (using the C function **strcpy()**) the corresponding string into the buffer. This is repeated with the mode of the motor by looking at the global **step_mode** variable, and then writing accordingly. The only difference here is that since buffer is already written into, we don't want to override the first string with the second one, so the C function **strcat()** is used instead. This *concatenates* the provided string onto the existing character array, rather than replacing it. It should also be mentioned that spaces follow the strings for direction and mode because the concatenation does not add any spacing, so it needs to be done on my end. Finally, this is repeated one more time with the **RPM** variable, and at this point the buffer is completed.

Here, the LCD is cleared, written to with the now-completed buffer character array, and then that same string is sent to the UART. The point of this so that each time a button is pressed, the mode of the motor is sent not only to the LCD, but also (via UART) to the PC, and displayed there as well.

With this function changed, the only other change I made was to the background loop, the **while(1)**. This is where the parsing of the user-input from the UART takes place. The reason all of these UART receive functions (which will be shown) are in the *background* as opposed to the foreground, is because we only need to parse the input when a full string is provided. Although we can run the task in the foreground, for example in an interrupt, all the current foreground tasks will operate as necessary without the need for the user's UART input. Without putting the character receive in a specific interrupt (like a DMA), we'd have to check the UART Tx periodically, which is not ideal as this can hypothetically lead to some inputs being missed (if their input is faster than the period at which it is being polled). By placing the **getstrU1()** function in a 'blocking' while loop until a full string is read, this ensures that no inputs will be missed (provided those routines with higher priority do not take too long). The code is shown below:

Listing 3: Main Program Loop

```
int main() {
  system_init();

  char input[15];
  char in_dir[3], in_mode[4];

  while (1) {
    while (!getstrU1(input, sizeof(input)));
    putsU1("\r\n");
    mCNIntEnable(FALSE);
    sscanf(input, "%s %s %d", in_dir, in_mode, &RPM);

    dir = (strcmp(in_dir, "CW")) ? CCW : CW;
    step_mode = (strcmp(in_mode, "HALF")) ? FS : HS;
   RPM = (RPM > 30) ? 30 : RPM;

    reset_clear_LCD();
    put_string_LCD(input);
    mCNIntEnable(TRUE);
  }

  return 0;
}
```

After initializing the system and all the peripherals, I declare three character arrays to be used as temporary buffers in the infinite loop. The first line within the **while(1)** is what I discussed above. This code constantly checks the UART buffer, and if a valid character is available then it places said character inside my provided character array (**input**). It returns a logical false so long as the buffer is neither full nor an escape character has been read. When the user presses enter (to send the string) the function returns true, and the parsing is complete.

With **input** now containing the string written on the Putty terminal on the PC, an empty line is written to improve readability on the terminal. Then, the change notice interrupt service routine is disabled. This is because the CN ISR that activates when a button is pressed accesses the LCD in order to write the mode corresponding to the buttons' states. *But*, the next section of code also interacts with the LCD to write what the user input. In order to avoid undesirable behavior resulting from both parts of code writing to the LCD at the same time, the CN ISR is disabled for a short while.

Next, the provided string is parsed into three variables. Using the C function **sscanf()**, the user input is placed into one buffer for the direction, one for the mode, and the global variable for the RPM of the motor. By comparing the text inside those two character arrays, the global variables for the direction and mode are then assigned accordingly. Since the lab handout said that the RPM of the motor should be capped at 30, a small bit of logic performs that limitation.

Finally, the LCD is cleared, and the provided string is placed there. Unlike the **decode_buttons()** function, there is no need to write to the UART the string. This is because the **getstrU1()** function actually echoes back the characters being written to the UART automatically (resulting in live-feedback). Now that all use of the shared resource (the LCD) is finished, the change notice interrupt is re-enabled.

Inside the communications file, **comm.c**, a function called **_mon_putc()** exists. Its existence might seem un-intuitive because it is not called at any point inside the file, but this function *overloads* the C function called **printf()** that allows for output to the console. The code for this function is structurally equivalent to **putcU1()** because it acts the exact same, except it redirects all console-output statements through UART1.

3 Testing and Verification

In order to test this project, I first added in all the LCD control code. I chose to add this first because we've already worked with the LCD using the **PMP** peripheral. I took the last LCD lab and compiled my code to ensure everything was copied correctly. After this, I added in the foreground ISR's used for all motor manipulation. This was also copying from past labs, but I once again compiled my code to ensure everything was copied correctly.

Because this code ran entirely in the foreground through the CN and timer ISR's, there was no interference of the two programs and both worked as expected. I then merged these two components in a more meaningful way, so the motor state was output to the LCD. Next, I copied in the **comm.c** and **comm.h** files, and then began adding each piece of code. This was essentially changing the hardware initialization function, and piece-by-piece adding the string manipulation code that parses the input from the UART. Since there was relatively little code written for this lab, it was very short to write this all and then test from there.

With all these pieces incorporated, the verification itself was quite quick and I simply just tested interacting with the motor through the terminal window, as well as the buttons. Both systems worked flawlessly, so after some minor changes to my LCD code (I had forgotten leading spaces in my substrings), the project was completed.

At first, testing just the LCD code was simple because I just had a test string placed on the LCD, which led to a very clear visual indicator of whether the system was working. Testing the motor was very easy as well, since it was all background based, so after pressing the buttons and seeing the motor's state change, that component was verified.

4 Conclusion

This lab served as an excellent introduction into asynchronous serial communication. Using the pre-built UART communication library removed the complexity of interacting with the UART, and made understanding how to actually work with the peripheral very direct.

Having now done one lab with serial communication and one with parallel communication, the relative advantages and disadvantages are quite clear. Obviously, serial communication has the inherent limitation of being slower. This is because the data is sent sequentially, one bit and then the next. Independent of whether or not there is a clock, this means that the data has an intrinsic speed limitation because larger data packets will take longer to transmit. Conversely, parallel communications (depending on the number of lines) can transmit multiple data bits at once, which is faster. There is also a downside to this speed, of course. In order to allow for simultaneous data transmission, there must be more hardware / data lines for the parallel communication which adds additional cost and size, and for a microcontroller with limited I/O, this can be a problem. And although this was not a factor for such a situation where the communication is happening on the same board or within one foot; parallel communication does have the possibility of becoming desynced, which can lead to loss of data. On the other hand, serial communications are always in-sync, as the transmission space is identical for each piece of data. One final disadvantage of parallel communication is that the protocol is only half-duplex, as opposed to the full duplex of many serial communique.

Because of these advantages and disadvantages of each type of communications, I think parallel communication should be used in short-range, high-speed systems. This is ideal because of the relatively lossy nature of serial systems, but also the simultaneous transfer of data allowing for higher transfer speeds. An example of this might be visual information (like monitors, TVs, etc.). On the other hand, I'd use serial protocols for longer-range systems, like sending small packets of data between two computers. This is because it less lossy, but has decreased transfer speeds.

This lab also demonstrates the importance of modular systems. This is because they allow for easily disabling and enabling systems when something is not functioning properly. In this case, modularizing the LCD and UART code meant that it was easy to implement the LCD functionality on its own, and then add the UART code on top. In the event that either system was misbehaving, this also allows for easier debugging.