**CMPE-250 Laboratory Exercise Six**

**Secure String I/O and Number Output**

By submitting this report, I attest that its contents are wholly my individual writing about this exercise and that they reflect the submitted code. I further acknowledge that permitted collaboration for this exercise consists only of discussions of concepts with course staff and fellow students; however, other than code provided by the instructor for this exercise, all code was developed by me.

Joel Yuhas

Performed March 9, 2016

Submitted March 10, 2016

Lab Section L2

Instructor: Muhammad Shaaban

TA: Peter Muller

Connor Goss

Stephen Moore

Lecture Section: 01

Professor: Alessandro Sarra

**Abstract**

Serial string input and output along with number output techniques were investigated and tested. These techniques were coded and tested on a Freescale Freedom KL46Z board using the KL46 universal asynchronous receiver/transmitter or UART. UART aided in preventing buffer overrun while the created program was being executed. This was done in order to explore the realm of subroutines for serial I/O of strings as well as investigating how to create the overarching programs which could use said subroutines. Once the subroutines and main code were created, they were run on the KL46 board and fed inputs via terminal window. The program would then branch to specific code when the proper command was entered. The primary function consisted of storing strings from the terminal, printing strings to the terminal, erasing stings, and printing the length of the string. The final program was able to produce correct results and the operation was a success.

**Procedure**

Several main subroutines had to be created that would allow the user to get/put strings to/from the terminal, which was the interface through which the user could input data. These subroutines included:

* GetStringSB
* LengthStringSB
* PutNumU
* PutStringSB

The mentioned subroutines were used in tandem with other subroutines that were developed in other exercises. The previous written subroutines included DIVU (exercise 2), PutChar and GetChar (exercise 5). These subroutines provided valuable operations that were necessary for the completion and implementation of the new subroutines.

*GetStringSB* was the subroutine responsible for reading a string from the terminal keyboard. The string which was being read in, would be written to an address that’s location would be stored in a variable called “String” (initialized in the beginning of the main code), which was taken in through register R0. Register R0 would house the address of the desired writing location while register R1 would specify the buffer capacity. The R1 register needed to be set prior to GetStringSB being called and typically was filled with a declared constant “MAX\_STRING” which in this scenario equaled 79. The buffer capacity was used in determining the max length an input string could reach..

The address housed in R0 was then moved to register R4, as register R0 would need to be used when executing GetChar. GetStringSB would then iterate through the terminal input one character at a time by using the GetChar subroutine to retrieve the value. The subroutine would then increment a counter located in register R3 (starting at 0) for each iteration that occurred. If the counter ever equaled the value of MAX\_STRING – 1 (explained later on) or if GetChar received the ASCII value for the “enter” key, then the subroutine would exit. Otherwise, GetStringSB would continue to loop until one of the two previously mentioned conditions were met, storing all characters into “String” along the way. GetStringSB pushed registers R0 – R4 and LR before the subroutine was run, and popped registers R0 – R4 and PC after. Pushing LR and popping PC provided the functionality of exiting the subroutine and returning back to where the subroutine was called, as BX LR command could not be used inside a subroutine which was called by another subroutine

*LengthStringSB* was the subroutine responsible for determining how many characters were in the inputted string. This was done by receiving the address for the inputted string through R0 and the buffer capacity in R1. LengthStringSB would then iterate through the string until it either reached the max buffer capacity or hit the null terminate. The null terminate was a special value located at the end of the stored string, in this case ‘0’, in order to signify that the string was over and the subroutine should quit. Each iteration would add one to a counter stored in R2, once the loop was done, the value located in R2 was equivalent to the number of characters in the string. LengthStringSB pushed registers R1, R3 and LR before the subroutine was run, and popped registers R1, R3 and PC after.

*PutNumU* was the subroutine that printed the text decimal representation of the unsigned word value in R0. This was done by receiving the *length* through register R0. PutNumU was different from the other subroutines in that it did not take in the buffer capacity. The length value was determined using the LengthStringSB subroutine, which needed to be gathered before PutNumU was ran. PutNumU would then transfer the length that was in R0 to R1 and would move the value of 10 into R0. This was done in order have the values properly inputted into the DIVU subroutine, because DIVU takes in R1 (the dividend) and R0 (the divider).DIVU was used to utilize a technique that allowed the length to be continually divided by 10 while storing the remainder at each step into a stack by “pushing” it. Once the length had been divided all the way through, the values would be “popped” off, out of the stack and then printed as the final, desired value. The stack system had to be used because the technique described would deliver the decimal value of the length, but in reverse order. In order to account for this, the stacks functionality of “last in first out” was utilized. The final output would then be printed to the terminal windows using GetChar. PutNumU pushed registers R0 – R2 and LR before the subroutine was run, and popped registers R0 – R2 and PC after.

*PutStringSB* was the subroutine responsible for displaying the string stored in memory on the terminal screen. This was done in a very similar fashion to GetStringSB, only instead of reading the values, it was writing the values. R0 would hold the address of the string while R1 would hold the buffer capacity. PutStringSB would then loop through the string continuously until either reaching the buffer capacity or hitting the null terminator. During each iteration, the subroutine would print each individual character to the terminal window by utilizing PutChar. PutStringSB pushed registers R0 – R3 and LR before the subroutine was run, and popped registers R0 – R3 and PC after.

Once the subroutines were created, the main code was written. This involved initializing UART to print a prompt on the terminal that would describe the options the user could pick from. The prompt “Type a string command (g, i, l, p):” was stored in a constant and loaded into a register. Before it could be loaded however, the constant needed to have a ‘0’ manually added to the end of the string to function as the null terminate.

After the initial prompt was printed, the program would receive the input from the user using GetChar. The program had to convert whatever value the user entered into lowercase. This was done by checking if the input was less than ‘Z’, which was converted to an ASCII value. If it was, that meant the value was an uppercase character and needed to be converted. Before it could be converted, a copy of the input was stored in R7, so that its original value (i.e. raw input data) could be stored in case it needed to be retrieved. If the input was uppercase, the code would branch and add 32 to the input. Adding 32 would change the inputs ASCII value, producing the lowercase equivalent. The, now lower case value, would then be put into store it in R7.

Then, the program would check to see if the inputted data was valid. Four different checks were created, one that checked if the data was equal to ‘g’, one for ‘i’, or for ‘l’ and one for ‘p’. This was the reason why the inputted data value was converted into lower case, so only four cases needed to be made instead of eight (to include the uppercase equivalent). If the data was equal to one of those values, that meant it was valid and could branch to the specific label, otherwise it would loop back and would wait for a new input variable.

The ‘g’ command was responsible for reading an entered string from the keyboard and then using GetStringSB to save the value. The main program would branch to the ‘g’ subroutine and, first, would print a ‘<’ and wait for the next user input. The subroutine would also load the “String” variable into R0, so that the contents could be stored to it using GetStringSB. The user would then type a string into the terminal and the program would take and store all characters until either “enter” was hit, the program reached the max buffer capacity -1 (to save room for the null terminate). Once the contents were successfully stored, CR and LF were put into the terminal so that the cursor could go down to the next line. LF (“0x0A”) and CR (“0x0D”) were values that, when put using PutChar, caused the terminal cursor to go down a line and start at the begging (all the way to the left) of the new one. This was so that the program would not continually print off of one line.

The ‘i’ command was responsible for loading in an empty string and saving it the “String” variable. This was done by simply loading a ‘0’ into R1, loading the “String” variable to R0, and then storing R1 into R0. CR and LF would then be executed with PutChar, moving the cursor to a fresh, newline.

The ‘l’ command was responsible for printing the length of the string. This was done by loading “String” into R0 and manually placing the buffer capacity into R1. LengthStringSB was then called, which would place the length into R2. A separate constant called “Length” was called. This held the string value of “*Length:* “which was stored into R0 and printed using PutStringSB. The reason for this was so that “Length: “would appear before the actual value of the length was printed. R2 was moved to R0 so that PutNumU could use it. PutNumU was called, which, using the previously mentioned method, would calculate the length of the string. After PutNumU had finished executing, CR and LF were put into the terminal so that the cursor would move down a line and to the left.

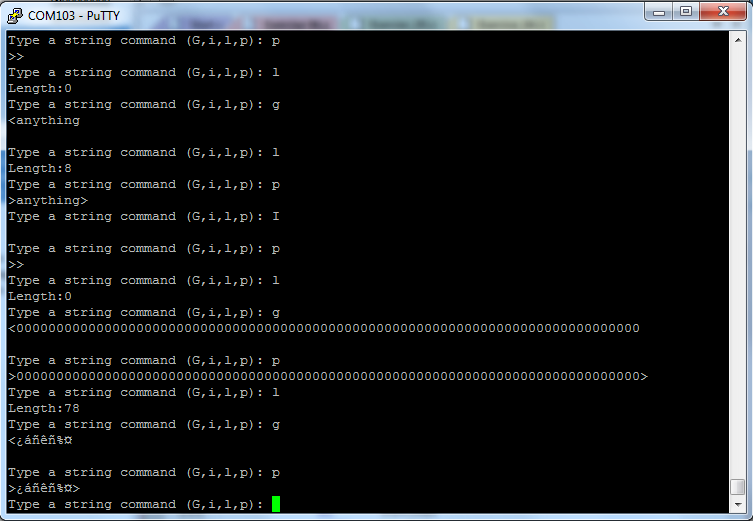
Finally, the ‘p’ command was responsible for printing the value stored in the “String” variable. ‘p’ would start off by printing a ‘>’ and then would load the “String” and buffer capacity using the same technique as all other commands and then call PutStringSB. This would print the string to the terminal. After PutStringSB was done executing, another ‘>’ was printed at the end, signifying that the string had finished printing. CR and LF were printed, again, and the cursor moved into its new positon.

Once any of these commands executed successfully, the program would then loop back, and wait for the next input after executing is applied code.

After all components had been implemented, the program was translated and built. This included adding a file called “Start.s” which was provided in Exercise 5. In order to actually run the program on the KL46 board, the board was connected to the computer and the terminal was opened via putty. The program was then run in the debugger which allowed it to work with the KL46.

**Results**

A number of different results were gathered during the exercise. Figure 1 shows a screen shot of the terminal window after it had successfully executed all necessary commands. Notice how the code is able to properly store keyboard data whenever ‘g’ is called. This is shown by printing the value in “String” whenever ‘p’ was called and after the string had been recorded. The strings length was also correctly calculated and shown after the command ‘l’ was called. Also notice how, when the string was empty, the program was able to properly report the correct size of ‘0’ and print the correct output. Clearing the string was also demonstrated by printing the value stored in the string prior to ‘i’ being called, and then printing the value stored in the string after, showing that the value had been erased. It should also be observed that the program was able to successfully convert values to lower case, which was demonstrated when ‘I’ was inputted. Not only did it run the code for ‘i’, but it printed the correct, upper case value ‘I’ in the terminal string. Finally, notice how when an attempt was made to overflow the buffer, the maximum size the program allowed the user to enter was 78, which was the correct results for this exercise (79 -1).



**Figure 1: Final Terminal Window Values**

Figure 2 shows a screen shot of the memory map produced by the program. A number of values needed to be recorded from the gathered memory map. These values included the memory ranges of the:

* Executable code in MyCodeAREA
* Constants (including all prompt and annotation strings)
* Variables (including the operational string)

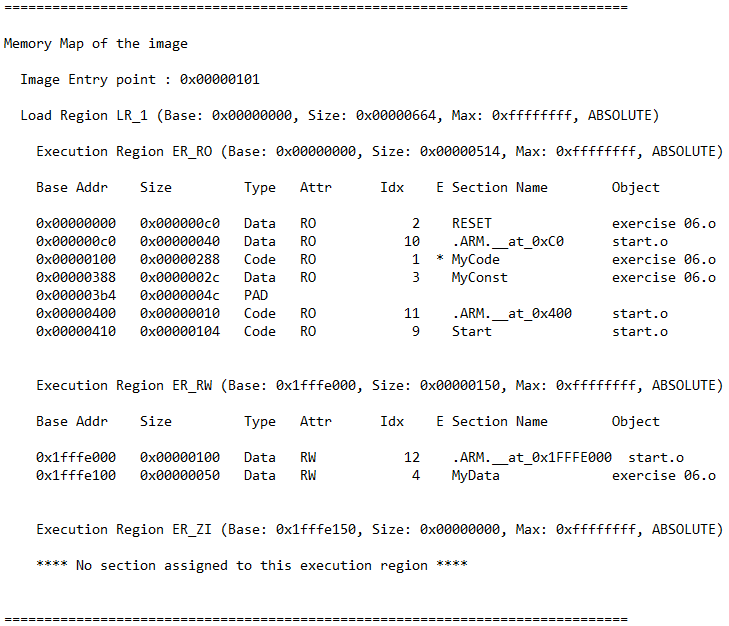
Those corresponding values were:

MyCodeAREA = 0x000000100 to 0x00000388

Constants = 0x000000388 to 0x000003B4

Variables = 0x1FFFE000 to 0x1FFFE150

As mentioned, Figure 2 shows the map where the results were gathered. These results were obtained by finding the base address of the desired location and then adding the size to that base address, which would give the subsequent range.



**Figure 2: Memory Map**

**Conclusion**

Serial string input and output along with number output techniques were investigated and tested. These techniques were coded and tested on a Freescale Freedom KL46Z board using the KL46 universal asynchronous receiver/transmitter or UART. Writing the code which lead to the execution of these task allowed for subroutines that dealt with serial I/O of strings to be studied and observed. This exercise also helped further develop a stronger understanding of writing and implementing code that would be run on a KL46. This included not only setting up the code using Keil in a specific way so the KL46 could read it properly, but also allowed for the user to practice connecting the KL46 and passing data to it via terminal. The exercise also allowed for the use of multiple subroutines, some of which had been created previously in other exercises. The results that were derived from the program, or the output that the program produced was what was expected and was considered a successful implementation. Overall, the exercise allowed for a further understanding of Keil, the KL46, subroutine implementation, and String I/O implementation alike. Some questions were posed by the exercise as well. Questions such as:

Why does the number of user-typed characters stored in the string have to be one fewer than the number of bytes allocated for the string? And,

Why does MAX\_STRING need to be defined as an EQUate rather than as a DCD in the MYConstAREA?

This was because:

The number of user-typed characters needed to have one fewer than the number of bytes allocated for the string because there needed to be room for the null terminate, or the ‘0’ placed at the end of the string. The null terminate was a special value that was used to tell when the string had ended and if it was not there could potentially lead to problems, such as having the code iterate past the end of the string into an area where there was no data.

MAX\_STRING needs to be defined as an EQUate because it allowed for its value to be moved into a register using MOVS instead of having to load its address to a register and then load its value. Ultimately, the MAX\_STRING could be saved as a constant, but saving it as an EQUate made it easier to use and more manageable to handle.

Overall, the exercise was a success and was able to accomplish all task that needed to be accomplished.