**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.

John Judge

Performed 3/3/16

Submitted 3/10/16

Lab Section 02

Instructor: Dr.Shaaban

TAs: Peter Muller

Stephen Moore

Connor Goss

Lecture Section 01

Professor: Allesandro Sarra

**Abstract**

This exercise investigated the serial string input and output and use of the Freescale Freedom KL465. The exercise’s objective was to create secure subroutines for serial I/O of strings by preventing buffer overrun. In the exercise four functions called *GetStringSB, LengthStringSB, PutNumU,* and *PutStringSB* were written and tested using a main function and other subroutines. The program written tested these functions by commands polled through the terminal on a separate machine with the operations taking place on the KL465 board.

**Procedure**

To begin the exercise four subroutines for secure polled serial I/O of strings needed to be written. These functions being *GetStringSB, LengthStringSB, PutNumU,* and *PutStringSB.* Each of these functions carried out a specific operation using secure polling between the terminal and the KL465 board. As a result, the *GetChar* and *PutChar* subroutines from a previous exercise were used to read characters and display characters, respectively, to the terminal. Once these subroutines were written a main function, as well as some other minor subroutines, were designed. Since each subroutine was branched too from other spots in the program it was essential that only the register specified input and output parameters would change. In order to achieve this, the registers that would change values were pushed and popped off the stack at the beginning and end of the subroutines, respectively. Once assembled the program tested the functionality of the four subroutines being tested.

The first of the four subroutines, *GetStringSB*, was designed so that when executed a string would be read from the terminal keyboard and stored in memory. The characters of the string would be stored starting at the address given by register R0 and the bytes of data could be stored up until buffer capacity given in register R1. Using a loop, the subroutine would call *GetChar* for each character typed on the terminal keyboard. Each character was echoed back to the terminal display using *PutChar* and the byte was stored in memory originating from address in R0. This loop continued until either the carriage return character was input by the user or the string reached one less than the buffer capacity. If the carriage return character was entered to the terminal keyboard, i.e. the user pressed enter, it signified the user terminating the string. When *GetChar* returned a carriage return character, or the buffer capacity was about to be reached, the string was null terminated by storing null in the next address in memory. Using this method, the string’s range could be easily identified as the area between the string pointer and the place in memory with null stored in it.

Since the string then had a defined range in memory the length of the string could be computed as the number of bytes stored between the string pointer and first null value. To compute this length *LengthStringSB* checked the byte stored at the string pointer. For each address that did not contain null, the address being checked would increment by one byte and a counter would increment by one. Once the first null value was found the subroutine returned the counter in Register R2. The value of the counter was therefore equal to the amount of bytes the string had stored.

The length of the string would eventually need to be displayed on the terminal display. In order to achieve this *PutNumU* would need to take the hexadecimal value from Register R2 and convert it to a decimal value and display the digits as characters to the terminal display. To preform this operation the subroutine *DivU* from a previous exercise was reused. By using *DivU* to divide the hexadecimal value of the length of the string by 10 the resulting quotient and remainder were the digits representing the length in decimal. By adding forty-eight to these values the corresponding ASCII value for the digits were found. Then *GetChar* could be used to display each of the digits to the terminal display.

In order to display a string to the terminal display the subroutine *PutStringSB* would need to display each character of the string, stored in memory, to the terminal display. This was achieved by looping through each byte in memory between the string pointer and the first null value in memory. For each byte stored in this range *PutChar* was called to display the character on the terminal display. This loop terminated when the first null byte was received or the buffer capacity had been reached. This subroutine would allow for any string to be read from memory so long as the string pointer was stored in R0 and the buffer capacity was stored in R1.

In order to test all of these functions a main program needed to be written to call them. When main program was written in such a way that when run it would prompt the user for input by displaying “*Type a string command(g,i,l,p):*” and waiting for input. For any of the letters (case non-specific) specified the character was echoed to the display and the terminal cursor was moved to the next line and it would direct the execution of one of the subroutines. For “*g*” the program would display “*<*” and then the user could input a string. While the string was input by the characters being typed on the terminal keyboard *GetStringSB* was called to store the string in memory and display the string on the terminal display. When the carriage return character was received the program looped to the beginning of the main function and prompt for input again on the next line in the terminal display.

If the next input received was *“i”* the string would be cleared from memory. This was achieved by storing null in the address of the string pointer. The same code used in this function was used to initialize the program by beginning with a black string. At the end of this function the program was looped back to the beginning of the main function.

When *“l”* was received the string’s length would be displayed following “*Length:*” The string containing the character “*Length:*” was displayed using *PutStringSB*. The decimal value of the length was displayed using the *LengthStringSB* subroutine. When this function was finished executing the program returned to the beginning of the main function.

If the next input was “*p*” the program would display the string stored in memory. The string would be displayed between two “*>*” characters. These characters were displayed using *PutChar* and the string itself was displayed using *PutStringSB*. Following this code’s execution the program would return to the beginning of the main function and request the used for input on the next line.

**Results**

The program was tested using several specified test cases. To begin, the input was *“l”* the results of this, and the following test cases, can be seen in Figure (1.1). Since the program responded with a string length of 0 it was concluded the program correctly initialized a blank string. The following three test cases required three different sized strings: an empty string, a string of length five, and a string over the buffer capacity of seventy-eight.

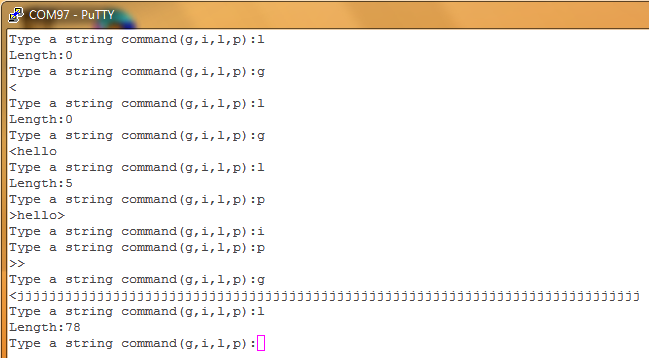


Figure (1.1): Results of the Test Cases

Inputting “g” and immediately pressing the “enter” button on the terminal keyboard tested the first string test case. This would also initialize an empty string. In order to test its functionality “*l*” was input. The program responded by correctly displaying a string length of zero.

The next case was tested by inputting *“g*” followed by the string “*hello*.” The program then stored the string in memory. The program was given the input “*l*” to check that it had been correctly stored. The program correctly responded with the length of five. Then *“p*” was input to display the string. The program correctly displayed the full string between the two greater-than symbols.

Before testing the next string the “*i”* command was tested. Once “*i”* was input then *“p”* was input. The program correctly displayed an empty string so it was concluded the program successfully cleared the string from memory.

Entering a string of over seventy-eight characters tested the final test case. The program successfully responded by responding with only a length of seventy-eight when “*l*” was input. During the exercise the buffer capacity was stored as an equate equal to seventy-nine so that the length of the string could easily be compared to the buffer capacity using the buffer capacity as an immediate. This would not be possible had the buffer capacity been used as a constant. The program, however, only allowed for strings of seventy-eight bytes to be stored as evident from this test case. That is because the seventy-ninth byte contains the null byte to signify the end of the string. Since the program successfully responded for each of the test cases so the overall functionality of the program was concluded to be correct.

From here the memory ranges of the program were analyzed. As seen in Figure (1.2) the memory map of the code displays the range, in memory, where the program executed. For this exercise the executable code in “*MyCode AREA*” was located between 0x00000100 and 0x00000264 in ROM. The constants used in the exercise we’re located between 0x00000364 and 0x00000390 in ROM. The variables, which were located in RAM, could be found between 0x1fffe100 and 0x1fffe23c. These ranges can be computed by using the base address and adding the value under the size column to determine the ending point.

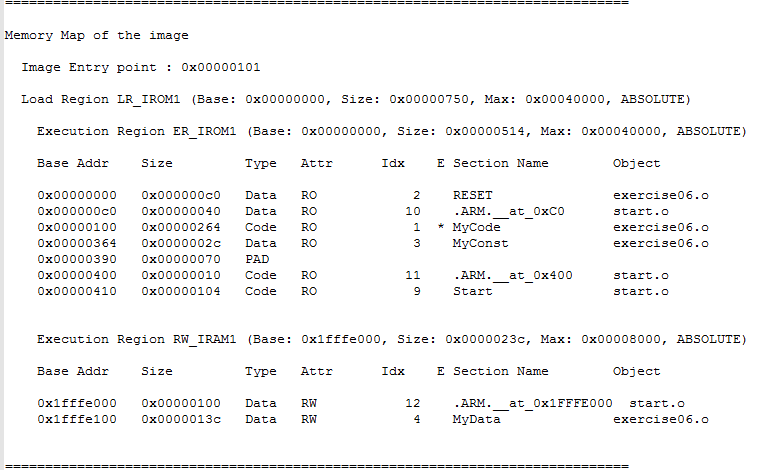


Figure (1.2): The Memory Map of the Program

**Conclusion**

This exercise investigated the serial string input and output and use of the Freescale Freedom KL465. The exercise’s objective was to create secure subroutines for serial I/O of strings by preventing buffer overrun. The program was written to test four subroutines called *GetStringSB, LengthStringSB, PutNumU,* and *PutStringSB.* When the required test cases were run the program responded with the correct results and the objectives of the exercise were met thus determining a successful exercise. The concepts of secure subroutines the UART of the Freescale Freedom KL465 could be implemented in future exercises. The subroutines written in this exercise will most definitely be utilized in future exercises using polling and strings. In the future the lessons learned during this exercise will be returned to and this exercise will be used as an example on how to correctly implement secure polling of strings.