**CMPE-250 Laboratory Exercise Eight**

**String Operations**

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 31,2016

Submitted April 7, 2016

Lab Section L2

Instructor: Muhammad Shaaban

TA: Peter Muller

Connor Goss

Stephen Moore

Lecture Section: 01

Professor: Alessandro Sarra

**Abstract**

String operations, specifically, string operations that ran on the Freescale Freedom KL46Z using universal asynchronous receiver/transmitter (UART0) were investigated and tested. Three subroutines were created, which would be used in a variety of ways that would manipulate the string in a desired fashion when called through a main program. These subroutines included “CopyString”, “ModifyString”, and “ReverseString”, all of which utilized specific techniques in assembly that stored and rearranged data in memory. The objective of this exercise was to create a collection of subroutines that could execute operations which would manipulate strings. These subroutines were implemented and tested through an overarching program which was written accordingly in order to accommodate the subroutines specific input/output requirements. Overall the program was able to produce the correct results and was ultimately a success.

**Procedure**

Before the main program and subsequent code could be written, three subroutines were created that would do the majority of the string manipulation. These subroutines included:

* CopyString
* ModifyString
* ReverseString

These subroutines were used in tandem with previously crafted subroutines that were developed in other exercises. The previous written subroutines used included PutChar and GetChar (exercise 5), GetStringSB (exercise 5), and PutStringSB (exercise5). These subroutines provided valuable operations that were necessary for the completion and implementation of the new subroutines.

*CopyString* was the subroutine responsible for doing, as the name describes, copying a string at a memeory address to another address. This was done by copying the characters from a null terminated source string from a specific address (in this case R0), and then placing them into a new memory address (R1). The CopyString subroutines main purpose was to take an initial value, a predetermined string defined in the code, and then copy that string into the memory address that the program would use to store the current string that would be manipulated. This was done by iterating over the string that needed to be copied one character at a time and then storing that character into the new address.

*ModifyString* was the subroutine responsible for manipulating a string in a very specific fashion. This involved taking a string stored at the memory address in R0 and then replacing:

* Every space character (0x20) with an underscore (0x5f)
* Every upper case character with its lower case equivalent
* Every number character (0 – 9) with a pound sign (#, 0x23)

The subroutine would modify these characters and store them in the same address, effectively overriding the previous string. This was done by iterating through the string and checking each character individually. If one of the characters was any of the cases described previously (space, uppercase, number), then the subroutine would jump to the code that would replace and restore the desired value.

*ReverseString* was the subroutine that, also as the name implied, reversed the string that was in the address stored in R0. This was achieved by having two counters, one that started at the beginning of the string (0), and one that started at the end of the string (string length – 1 for the null pointer). The counters would load their respective characters into two different registers and then store them back into the string in the opposite location locations.

All of these strings did not modify any registers, but simply modified the memory from which the addresses pointed to.

Once the subroutines were created, the main code was written. This involved initializing UART and printing a prompt on the terminal that would describe the options the user could pick from. The prompt “Type a string command (g, h, m, p, r):” was stored in a constant and loaded into the register R0, where PutStringSB would print it. Before it could be loaded however, the null terminate needed to be added at the end of the string, which would allow the PutStringSB subroutine to exit once it reached it.

Next, before the string could be printed out, a separate string with the value “initial string” was loaded into R1. This, along with the address of the string that the program would use to store the value of the user inputted/modifiable string would be stored in R0. Both these registers would be utilized when CopyString subroutine was called immediately before the main loop ran. This step would load the initial value “initial string” into the address in R0, which meant that “initial string” would be the string that would be loaded by the program and be the string which would be modifiable until overwritten by another string.

After the initial prompt was printed, the program would receive the input from the user using GetChar. If the input was uppercase, the program had to convert whatever value was entered into lowercase. This was done by checking if the input was less than ‘Z’, which was converted to an ASCII value. If the value was less than ‘Z’, that meant it was potentially an uppercase value and needed to be converted. The code would branch off and then check to see if the value was above ‘A’. If it passed both of those requirements, that meant that it was an uppercase letter. The letter would then be converted into its lowercase equivalent. This was done so that both upper case and lower case inputs would be able to be read by the program while only having to make triggers for the lowercase scenarios.

This was done by taking a copy of the input received from the terminal keyboard and storing it in R7, so that its original value could be saved in case it needed to be retrieved. Then, the code would add 32 to the input, changing its ASCII value so that it produced the lowercase equivalent, and stored it back in R0. The code would then break back and continue with the rest of the main code

The program would also check to see if the inputted data was the “enter” key. If it the inputted value was the enter key, then the program would skip a line, reprint the initial prompt, and start over.

At this point, if the inputted value was not an uppercase letter or the enter key, the program would check to see if any of the key commands had been inputted. These commands being

* g, which would get the operational string from the terminal keyboard,
* h, which would print out a list of valid commands
* m, which would execute the modify subroutine
* p, which would print the operational string on the terminal string
* r, which would execute the reverse string

The ‘g’ command mainly utilized the GetStringSB subroutine, which did the work of reading user inputted string values entered in on the keyboard and ending when the user hit enter. The g command also was responsible for creating an environment in which the GetStringSB command could operate properly. This meant loading the memeory address of where the string would be stored into R0, and then loading the max string size “MAX\_STRING” into R1. Once those two conditions had been met, GetStringSB could be called with no further issue. All other commands followed the same format, in terms of loading the String location and the MAX\_STRING. Once command was done, it would loop back to the initial loop and wait for another command to be inputted.

The ‘h’ command was responsible for printing out a help tab. This simply listed all of the commands as well as a one-word description of said command:

* “g (get), h (help), m (modify), p (print), r (reverse)”

This was done by loading the address of the string which held the above contents and then printed them out on the terminal screen using PutStringSB. Once that had been done successfully, the command would go to the next line and loop back to the beginning of the main loop.

The ‘m’ command was responsible for creating an environment which would successfully call the ModifyString subroutine. This was done by loading in the string address into R0 and calling StingModify. After the command had finished, it would branch to the P command, where it would print out the contents of the string and then branch back to the main loop.

The ‘p’ command was responsible for printing the current string that was stored in memory. This command mainly consisted of calling PutStringSB, which did all of the heavy lifting by printing the string onto the terminal window. Aside from that, the command also performed the utility duties such as properly loading the address into R0 and loading the MAX\_STRING value into R1. After the command was completed, it would branch back to the main loop and wait for the user to input another command.

Finally, the ‘r’ command was responsible for reversing the string stored in memeory. This was another simple command, which called the StringReversal subroutine, which did all of the work and then branched to the print command in order to display the new contents.

All of these commands called a utility subroutine known as “correct” before they were executed. The only operation this subroutine did was move the pointer to the new line and branch back to the command. These commands also all called the address where the modified string was stored (variable “String”) into R0, as well as calling the MAX\_STRING into R1 before any functions were done. This is what was meant by “creating the appropriate environment” from which the subroutines would work correctly, because had these commands not been there, then the program would potentially fail.

If any of these commands were not called, then the program would print out “Invalid command” and branch back to the beginning of the main loop.

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. Then the program was executed and ran in the debugger and the programs functionality was tested.

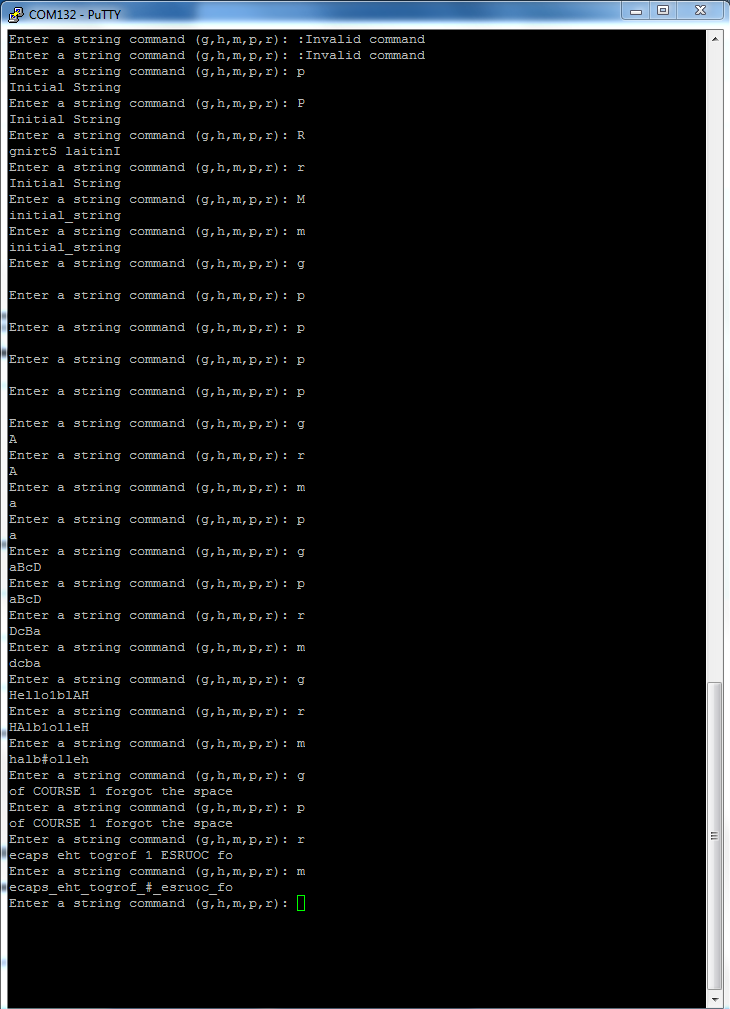
Once all test had proven successful, a screen capture of the terminal window was taken and saved and the project was considered complete.

**Results**

Two specific forms of results were gathered. Figure 1 shows the screen capture of the terminal windows. This screen capture contains within it one or more of the following scenarios:

* Uppercase and lowercase commands
* Various other commands before g or G command
* G or g command with input string of no characters (enter pressed immediately)
* Various other commands after the g or G command
* G or g command with input string
* Various other commands after the g or G command

Along with these scenarios, all of the commands were called at some point. This was done in order to ensure that the program produced the correct functionality and to check to see if its outputs were expected.



**Figure 1: Final Terminal Window Values**

The second batch of data involved using 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)
* RAM used (including the operational string)

Those corresponding values were:

MyCodeAREA = 0x000000410 to 0x0000072F

Constants = 0x000000204 to 0x000002D7

RAM = 0x1FFFE000 to 0x1FFFE1A0

What these memory ranges show are the exact areas in which the code is being stored and drawn from. For example, the MyCode section displays the areas in memory where the physical code is being stored and read from, while Constants and RAM show the same thing. This is useful information because while the program is running, these memory locations can be looked up and the values which are stored in them can be deciphered. Knowing these memeory locations is also useful because the size of the program can be effectively gauged.

**Conclusion**

String operations were investigated, specifically, string operations that ran on the Freescale Freedom KL46Z using universal asynchronous receiver/transmitter (UART0). The functionality of the string operations that were under investigation resided in three subroutines. These three subroutines were created using a variety of ways that would produce the desired string manipulations. These subroutines, which included “CopyString”, “ModifyString”, and “ReverseString”, worked alongside a main program that would check for certain commands entered in by the user, and execute the desired operations. The objective of this exercise was to create a collection of subroutines that could execute operations which would manipulate strings which was done successfully. The three subroutines, along with the main program, were implemented in a way that produced expected results, and were able to store, manipulate, and produce strings whilst following constraints that were laid out in the exercise guild lines. Older subroutines were used from previous exercise and compiled together to create the final code. Overall, the exercise was a success and was able to accomplish all task that needed to be accomplished.