Laboratory Assignment #3

Autumn 2013 TCES 430 – Microprocessor System Design

by

Alvin Baldemeca and Edward Bassan

Date: 10/29/2013

Table of Contents

I. Objective

The purpose of this lab is to familiarize ourselves with loading data on to program memory on the pic micro-controller. We are asked to create a program in C, that will retrieve data stored in program memory and move it into data memory. We are to simulate the program using MPLAB SIM. The lab specifies that we create a program that has two subroutines, the first subroutine retrieves the string stored in program memory and the second subroutine counts the number of characters in the word. A static access location for the string retrieved and the count must be used. Finally, we must create another program which modifies the first program so that it only has one subroutine instead of two.

II. Procedure

We created a new project for the MPLABX IDE, by choosing a standalone project and the device we selected is the PIC18F4520. For our hardware tools we selected the simulator, finally we selected the XC8 compiler. In the XC8 compiler user guide declaring something as a constant using the declaration "const" load the data into program memory. We created five words with varying length like the example below:

```
const char STRING1[12] = "Hello World\0";
const char* STRING2[15] = "This is a test";
const char* STRING3 = "Hi there";
const char* STRING4 = "Alvin";
const char* STRING5 = "Edward";
```

The code for Lab3.c and Lab3_mod.c is included in Appendix A of this lab report. To print anything to the UART 1 Output so that we can see it on the MPLABX IDE we had to add the header file 18F4520_g.lkr, include the a function putch in our code and also added the SPEN and TXEN (set to 1) in the main function (see code snippet below).

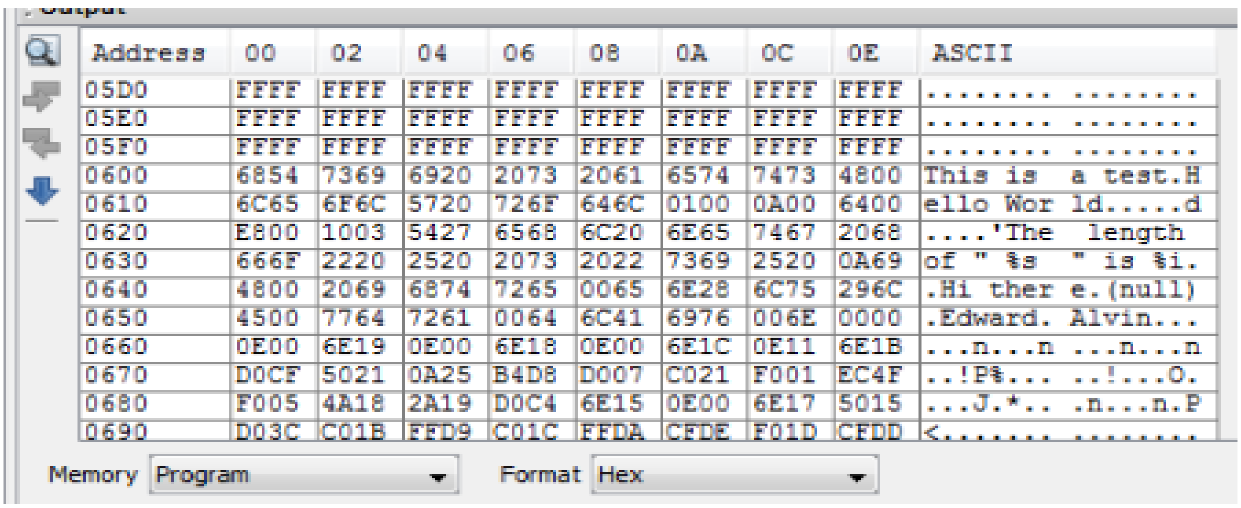
```
#include <stdio.h>
#include <stdlib.h>
#include <p18f4520.h>
#pragma config WDT=OFF

void putch(char c)
{
   while(!TRMT);
   TXREG = c;
}
```

```c
int main(int argc, char** argv){

   SPEN = 1;
   TXEN = 1;
   printf("Hello World!\n");
   while(1);
   return (EXIT_SUCCESS);
}
```

Once the code was written we compiled and ran the code. We viewed the data stored in program memory by going in to the “Windows” menu on the MPLABX IDE. The full menu path is Windows > PIC Memory Views > Program memory, we then had to change the “Format” option to Hex in order to view the ASCII characters. Figure 1 shows some of the data that can be viewed in program memory.



Address	00	02	04	06	08	0A	0C	0E	ASCII
05D0	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	....
05E0	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	....
05F0	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	....
0600	6854	7369	6920	2073	2061	6574	7473	4800	This is a test.H
0610	6C65	6F6C	5720	726F	646C	0100	0A00	6400	ello Wor ld.....d
0620	E800	1003	5427	6568	6C20	6E65	7467	2068	....'The length
0630	666F	2220	2520	2073	2022	7369	2520	0A69	of " %s " is %i.
0640	4800	2069	6874	7265	0065	6E28	6C75	296C	.Hi ther e.(null)
0650	4500	7764	7261	0064	6C41	6976	006E	0000	.Edward. Alvin...
0660	0E00	6E19	0E00	6E18	0E00	6E1C	0E11	6E1B	...n...n ...n...n
0670	D0CF	5021	0A25	B4D8	D007	C021	F001	EC4F	..!P%... ..!...O.
0680	F005	4A18	2A19	D0C4	6E15	0E00	6E17	5015	...J.*.. .n...n.P
0690	D03C	C01B	FFD9	C01C	FFDA	CFDE	F01D	CFDD	<....... ....

Figure 1. Program Memory (MPLABX IDE View)

III. Data and Analysis

From Figure 1 above the ASCII column contains the strings that we stored into program memory. We declared our strings 3 different ways the first way was counting the letters in the string plus the null character “\0” and storing it in a char array (char[ ]) :

```c
const char STRING1[12] = "Hello World\0";
```

This worked and is probably a good way to store a string of known length to memory. The second method we tried is using the following:

```c
const char* STRING2[15] = “This is a test";
```

as we can see from Figure 1 this works. Note that we did not include a terminating null character. The length of the string is 14. So did the compiler insert a null character? Although from what we could see 00 is inserted at address 060F (see Figure 1. Note: The word contains 4800 which represents “H” and a null ascii character.) we should not rely on this from the compiler which might cause some bugs when the PIC programmer actually flashes this into the PIC18F4520.

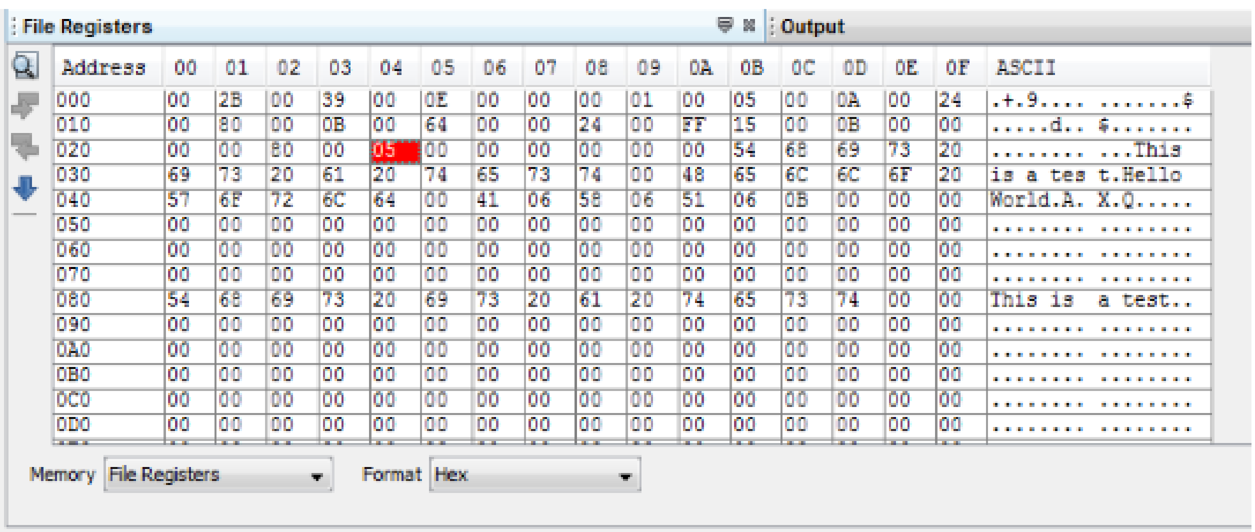
The second method to declare the string was attempted with varying size of array, if the array for example was changed to size 14 (STRING2[14]) we get a compiler warning and when we ran this code the printed string kept printing ascii characters until it came across a null character. In this case “This is a testHello World” was printed instead fo just “This is a test”.

The third method which is the best of the three that we tried is declaring the string as follows:

```
const char* STRING3 = "Hi there";
const char* STRING4 = "Alvin";
const char* STRING5 = "Edward";
```

The compiler correctly stores this into program memory with a terminating null character which is one method found on user guide for the XC8 compiler.

Looking at the file register we observed that the first two methods we used actually moves the whole string into data memory as can be seen in Figure 2. We can also observe from the assembly code in Figure 3 that the assembly code uses tables (TBLDR) to move the strings into data memory.



Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
000	00	2B	00	39	00	0E	00	00	00	01	00	05	00	0A	00	24	.+.9.... ....$
010	00	80	00	0B	00	64	00	00	24	00	FF	15	00	0B	00	00	.....d.. $.......
020	00	00	80	00	05	00	00	00	00	00	00	54	68	69	73	20	....This
030	69	73	20	61	20	74	65	73	74	00	48	65	6C	6C	6F	20	is a tes t.Hello
040	57	6F	72	6C	64	00	41	06	58	06	51	06	0B	00	00	00	World.A. X.Q.....
050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
080	54	68	69	73	20	69	73	20	61	20	74	65	73	74	00	00	This is a test..
090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

Figure 2. Data Memory (MPLABX IDE View)

The third string “Hi there” is not seen, when we ran the debugger, the program actually loads ascii A (41 hex at address 046) for “Hi there”, X for “Alvin” and Q for “Edward”.

```
!     const char STRING1[12] = "Hello World\0";
0x834: MOVLW 0xF
0x836: MOVWF TBLPTRL, ACCESS
0x838: MOVLW 0x6
0x83A: MOVWF TBLPTRH, ACCESS
0x83C: LFSR 2, 0x3A
0x83E: NOP
0x840: MOVLW 0xC
0x842: TBLRD*+
0x844: MOVFF TABLAT, POSTINC2
0x846: NOP
0x848: DECFSZ WREG, F, ACCESS
0x84A: BRA 0x842
!     const char STRING2[15] = "This is a test";
0x84C: MOVLW 0x0
0x84E: MOVWF TBLPTRL, ACCESS
0x850: MOVLW 0x6
0x852: MOVWF TBLPTRH, ACCESS
0x854: LFSR 2, 0x2B
0x856: NOP
0x858: MOVLW 0xF
0x85A: TBLRD*+
0x85C: MOVFF TABLAT, POSTINC2
0x85E: NOP
0x860: DECFSZ WREG, F, ACCESS
0x862: BRA 0x85A
!     const char* STRING3 = "Hi there";
0x864: MOVLW 0x6
0x866: MOVWF 0x47, ACCESS
0x868: MOVLW 0x41
0x86A: MOVWF STRING3, ACCESS
!     const char* STRING4 = "Alvin";
0x86C: MOVLW 0x6
0x86E: MOVWF 0x49, ACCESS
0x870: MOVLW 0x58
0x872: MOVWF STRING4, ACCESS
!     const char* STRING5 = "Edward";
0x874: MOVLW 0x6
0x876: MOVWF 0x4B, ACCESS
0x878: MOVLW 0x51
0x87A: MOVWF STRING5, ACCESS
```

Figure 3. Lab3.c compiler assembly code

Figure 3, the assemly code associated with _const char* STRING3 = “Hi there”_ we can see that we load 0x06 into data memory x047 and 0x41 (‘A’) to data memory associated the label STRING3 (0x46 data address). The last three strings are only stored as a pointers compared to the first two which is moved from program memory to data memory upon initialization. The static char buffer[ ] in Lab3.c starts at address 080 in the data memory, we observed this changing to the different strings that we try to move to the buffer. The length of the string is found on data address 024 which again was observed to change to the proper string length for a given string.

The C code it self was trivial we moved the string into the buffer by using the following C statements below also see Appendix B for the flowcharts for the subroutines :

```
while(progString[i] != 0)
 {
    buffer[i] = progString[i];
    i++;
 }
```

The assembly code snippet for this part of the code does something with TBL (tables) add FSR (pointers). For the first two strings this seems like a wasted effort since the string is already loaded in to data memory (see Figure 2) so we don't need to move it to our buffer, we can just print it to the UART. The last three strings however is not in data memory so we do have to move this into data memory, in our case it gets moved into the buffer.

```
!     {
!         buffer[i] = progString[i];
0x914: MOVFF __pcstackCOMRAM, divisor
0x916: NOP
0x918: MOVFF 0x2, 0x4
0x91A: NOP
0x91C: MOVF i, W, ACCESS
0x91E: ADDWF divisor, F, ACCESS
0x920: MOVF quotient, W, ACCESS
0x922: ADDWFC 0x4, F, ACCESS
0x924: MOVFF divisor, TBLPTRL
0x926: NOP
0x928: MOVFF 0x4, TBLPTRH
0x92A: NOP
0x92C: CLRF TBLPTRU, ACCESS
0x92E: MOVLB 0x0
0x930: MOVLW 0x80
0x932: ADDWF i, W, ACCESS
0x934: MOVWF FSR2L, ACCESS
```

Figure 4. Lab3.c compiler assembly code (Indirect addressing using tables and pointers)

We looked at the subroutine and as we expected the assembly code uses the "CALL" assembly opcode which uses the PIC18F4520 stack. Figure 5 shows how the XC8 compiler translate our C code subroutine into assembly.

```
!
!          retrieve_word(STRING1);
0x87C: MOVLW 0x0
0x87E: MOVWF 0x2, ACCESS
0x880: MOVLW 0x3A
0x882: MOVWF __pcstackCOMRAM, ACCESS
0x884: CALL 0x90A, 0
0x886: NOP
!          count_and_print(buffer);
0x888: MOVLB 0x0
0x88A: MOVLW 0x0
0x88C: MOVWF 0x23, ACCESS
0x88E: MOVLB 0x0
0x890: MOVLW 0x80
0x892: MOVWF string, ACCESS
0x894: CALL 0x9F8, 0
0x896: NOP
```

Figure 5. Lab3.c compiler assembly code (Subroutine)

IV. Conclusion

We got both our C programs to compile and run on the MPLAB sim. The behavior of the program is what we expected. We learned how the XC8 compiler translate subroutines in C to assembly using CALL which uses the stack. Analyzing the assembly code that XC8 generates we learned how different strings stored into program memory is moved into data memory and also how pointers to the data in program memory is moved in to the data registers. Finally, we learned how function declaration, subroutine and passing pointers in C works in assembly.

Appendix A Code for Lab3.c

```
/*
 * File:   Lab3_mod.c
 * Author: Alvin Baldemeca, Edward Bassan
 * UWT TCES 430
 * Lab 3
 * Prof. Sheng
 * Created on October 22, 2013, 11:28 AM
 */

#include <stdio.h>
#include <stdlib.h>
#include <xc.h>
#include <p18f4520.h>
#pragma config WDT=OFF
#define MAX_LENGTH 64

static char buffer[MAX_LENGTH];
static int length;

void putch(char c)
{
   while(!TRMT);
   TXREG = c;
}

/**
 * This function stores a string into a char[64] called buffer.
 * @param progString the pointer to the string or char* to load in to the buffer
 */
void retrieve_word(const char* progString){

   int i = 0;
   while(progString[i] != 0)
   {
      buffer[i] = progString[i];
      i++;
   }
   buffer[i] = 0;
```

```
}

/**
 * This function prints out to the UART I/O a message containg the word to be
 * printed out and the length of the word
 * @param  -string the char pointer to the string or char[0]
 * @return - returns 0 if printf was successfully executed.
 */
int count_and_print(char* string)
{
    int i = 0;
    while(string[i]!= 0)
    {
        i++;
    }
    length = i;
    return printf("The length of \" %s \" is %i\n", string, length);
}

/**
 * The main function of the program calls on two helper functions.  One function
 * retrives a string and stores it into a buffer.  The other function counts the
 * number of characters in the word and prints out the word.
 * @param argc -the number of command line arguments (not used)
 * @param argv -the pointer to the command line arguments (not used)
 * @return -0 on success
 */
int main(int argc, char** argv) {

    SPEN = 1;
    TXEN = 1;
    const char STRING1[12] = "Hello World\0";
    const char STRING2[15] = "This is a test";
    const char* STRING3 = "Hi there";
    const char* STRING4 = "Alvin";
    const char* STRING5 = "Edward";

        retrieve_word(STRING1);
        count_and_print(buffer);

        retrieve_word(STRING2);
        count_and_print(buffer);
```

```
        retrieve_word(STRING3);
        count_and_print(buffer);

        retrieve_word(STRING4);
        count_and_print(buffer);

        retrieve_word(STRING5);
        count_and_print(buffer);

    while(1);
    return (EXIT_SUCCESS);
}
```

Code for Lab3_mod.c

```
/*
 * File:   Lab3_mod.c
 * Author: Alvin Baldemeca, Edward Bassan
 * UWT TCES 430
 * Lab 3
 * Prof. Sheng
 * Created on October 22, 2013, 11:28 AM
 */

#include <stdio.h>
#include <stdlib.h>
#include <xc.h>
#include <p18f4520.h>
#pragma config WDT=OFF
#define MAX_LENGTH 64

static char buffer[MAX_LENGTH];
static int length;

void putch(char c)
{
   while(!TRMT);
   TXREG = c;
}

/**
 * This is a helper function for the main function.  It is a subroutine that
 * retrieves the string stored in the program memory and loads it into data
 * memory and counts how many character/s the word is.
 * @param progString the pointer to the program string
 */
void get_count_print(const char* progString){

   int i = 0;
   while(progString[i] != 0)
   {
      buffer[i] = progString[i];
      i++;
   }
   buffer[i] = 0;
```

```
    length = i;
    printf("The length of the word \" %s \" is %i\n", buffer, length);
}

/**
 * The main function calls a subroutine.  The subroutine places each char in the
 * string in a buffer, it counts the number of characters and prints the word
 * and the number of characters in the word.
 * @param argc the number of command line arguments (not used)
 * @param argv the pointer to the command line arguments (not used)
 * @return
 */
int main(int argc, char** argv) {

    SPEN = 1;
    TXEN = 1;
    const char* STRING1 = "Hello World";
    const char* STRING2 = "This is a test";
    const char* STRING3 = "Hi there";
    const char* STRING4 = "Alvin";
    const char* STRING5 = "Edward";

    get_count_print(STRING1);
    get_count_print(STRING2);
    get_count_print(STRING3);
    get_count_print(STRING4);
    get_count_print(STRING5);

    /*
     * This while loop is needed for the simulator because so that it does not
     * loop indefinitely
     */
    while(1);
    return (EXIT_SUCCESS);
}
```

Appendix B. Flowcharts for subroutine