Zachary Moshansky

Project 0

Lab L2C – C4

University of British Columbia

EECE 315

Thursday, February 07, 2013

Table of Contents

[List of Figures 2](#_Toc348129169)

[1.0 Introduction 1](#_Toc348129170)

[2.0 Solution Approach 1](#_Toc348129171)

[2.1 Part 1 - IEEE floating point format 1](#_Toc348129172)

[2.2 Parts 2 & 3 – MS-DOS 1](#_Toc348129173)

[2.2.1 Part 2 – Capitalize and Reverse String 1](#_Toc348129174)

[2.2.2 Part 3-1 – Randomize color of screen 1](#_Toc348129175)

[2.2.3 Part 3-2 – Output String to any part of screen 1](#_Toc348129176)

[2.3 References 2](#_Toc348129177)

[3.0 – Diagrams 2](#_Toc348129178)

[4.0 Implementation 3](#_Toc348129179)

[4.1 Part 1 - IEEE floating point format 3](#_Toc348129180)

[4.2 Part 2 – Capitalize and Reverse String 3](#_Toc348129181)

[4.3.1 Part 3-1 – Randomize color of screen 4](#_Toc348129182)

[4.3.2 Part 3-2 – Output String to any part of screen 4](#_Toc348129183)

[5.0 Results & Test Cases 5](#_Toc348129184)

[6.0 Testing 8](#_Toc348129185)

[7.0 Comments & Conclusion: 8](#_Toc348129186)

[8.0 Appendix 9](#_Toc348129187)

[8.1 Part 1 Code – “315P0AS.c” 9](#_Toc348129188)

[8.2 Part 2 Code – “part2\_debug.asm” 10](#_Toc348129189)

[8.3 Part 3 Code – “part3\_debug.asm” 11](#_Toc348129190)

# List of Figures

[Figure 1 - Part 2 Results 6](#_Toc348122069)

[Figure 2 - Part 3-1 Results \*\*New-line output after program exits and therefore not “dazzled.” 7](#_Toc348122070)

[Figure 3 - Part 3-2 Results 7](#_Toc348122071)

# 1.0 Introduction

The purpose of this project was to gain familiarity with how x86 Windows and Linux computers stored IEEE 754 (Floating Point) numbers. Furthermore, it was intended that students would learn about system calls and assembly programming. Learning how different platforms represent data is integral as the distinctions between them can have an enormous impact on a system if it is migrated from its original hardware/operating system as well as in cross-platform applications where interoperability between different platforms is required. Learning about system calls is necessary in broadening students’ understanding of how low level operating system and hardware interactions are performed as well as gaining a background that is necessary to understand how the boot process and kernel operate. Finally, assembly programming teaches valuable information about the exact operation of low level instructions and is useful for truly understanding how computers and software operate.

# 2.0 Solution Approach

## 2.1 Part 1 - IEEE floating point format

I approached by using Visual Studio to create the C application. The approach I took, was to take the user input, store it in both a single and double precision floating point format, and then iterate through each byte of the stored data by using an unsigned char pointer set to the first byte of the data structure and incrementing the address and iterating until I had output all bytes of the data structure.

## 2.2 Parts 2 & 3 – MS-DOS

I used DEBUG running under DOSBOX; to facilitate easier typing of the code I used Notepad++ outside of the virtual machine then typed the code using DEBUG. To reduce effort, I wrote the hex file out of debug then edited in a hex editor as needed and reloaded into DEBUG.

## 2.2.1 Part 2 – Capitalize and Reverse String

I solved part 2 by writing the input to a buffer, then using the start and offset to calculate the index of the last character entered. Then I checked if ASCII value was lower case and if so uppercased it before writing it to screen. I then repeated this as I iterated through the entire buffer from end to start.

## 2.2.2 Part 3-1 – Randomize color of screen

I XOR’d the attribute byte with the ASCII byte and then carried it forward, using it as a seed to repeat the operation for the next position, achieving pseudo randomness.

## 2.2.3 Part 3-2 – Output String to any part of screen

I started by setting up the direction flag which was necessary for the “movsb” instruction. Then I setup ES:DI and DS:SI, by calculating the necessary offset to achieve the proper position, so that I could utilize the “movsb” instruction to copy from the string [buffer] to the ASCII buffer. After moving the ASCII character I pushed the String Index (SI) to the stack and copied the character into a temporary buffer so that I could copy the attribute byte into the address pointed to by SI so I could utilize “movsb”. Finally I popped SI from the stack and restored the temporarily buffered character, that would have been overwritten by the attribute data, and checked if it contained the terminating character “$”, before looping again.

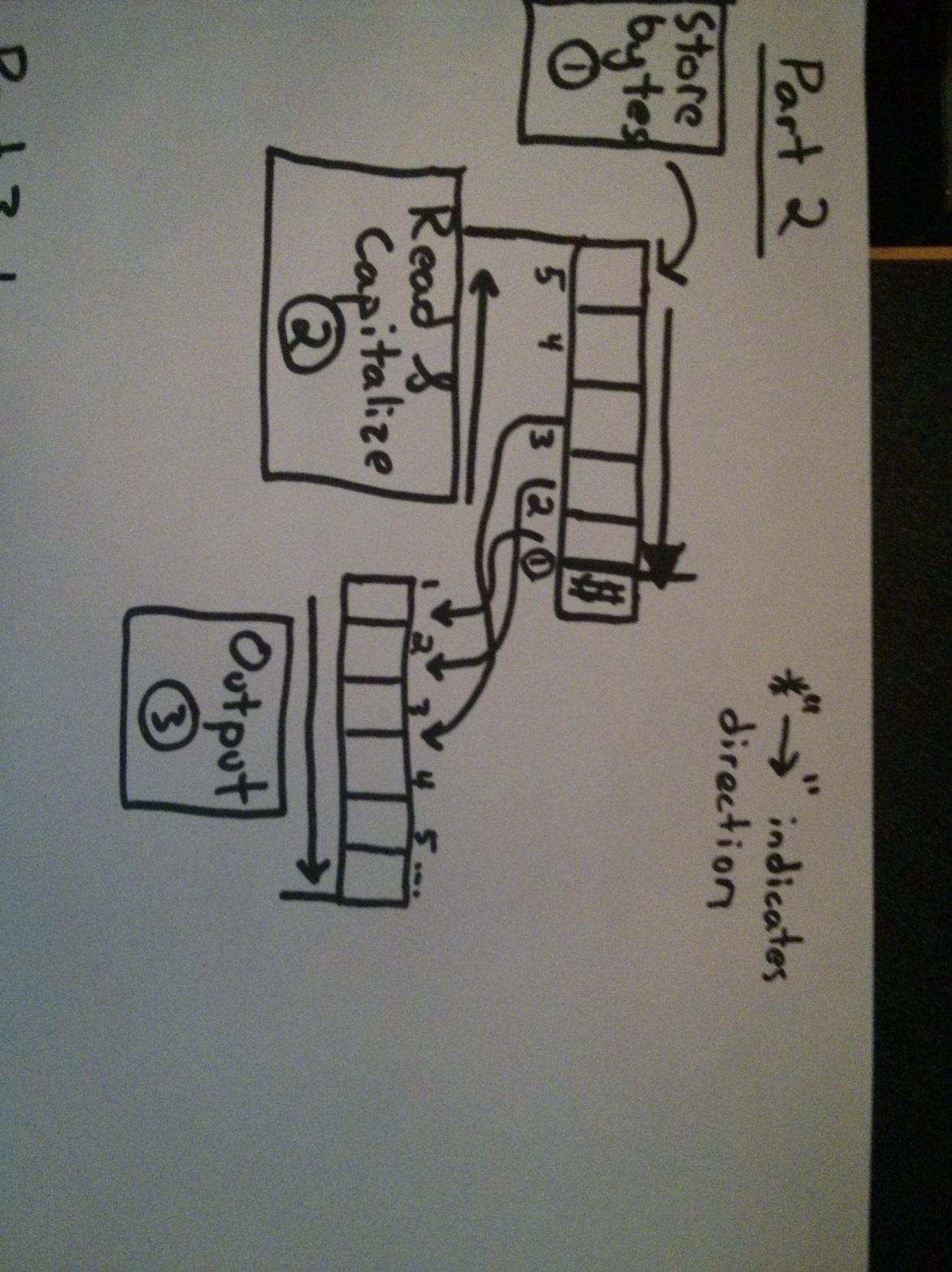
## 2.3 References

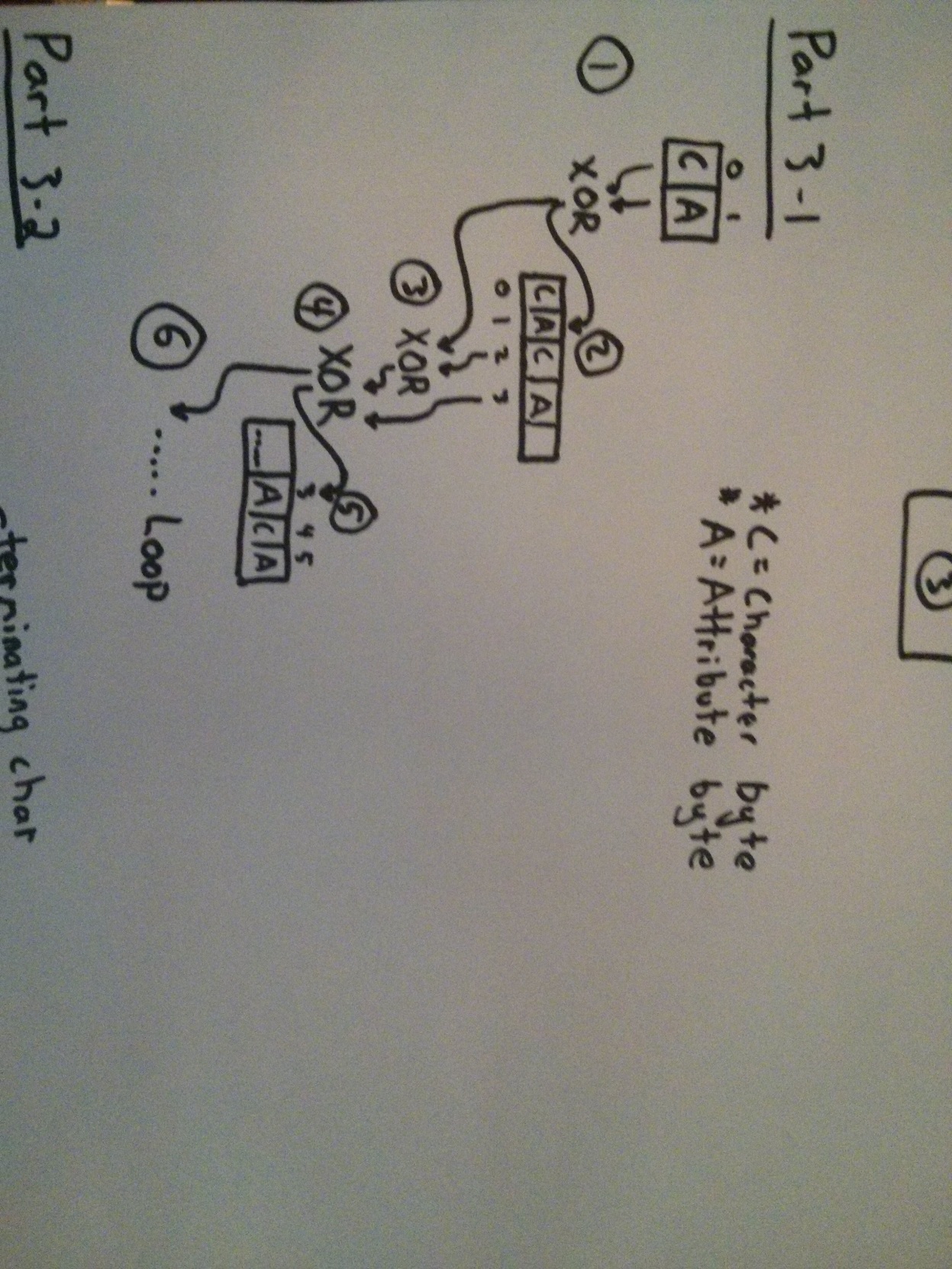
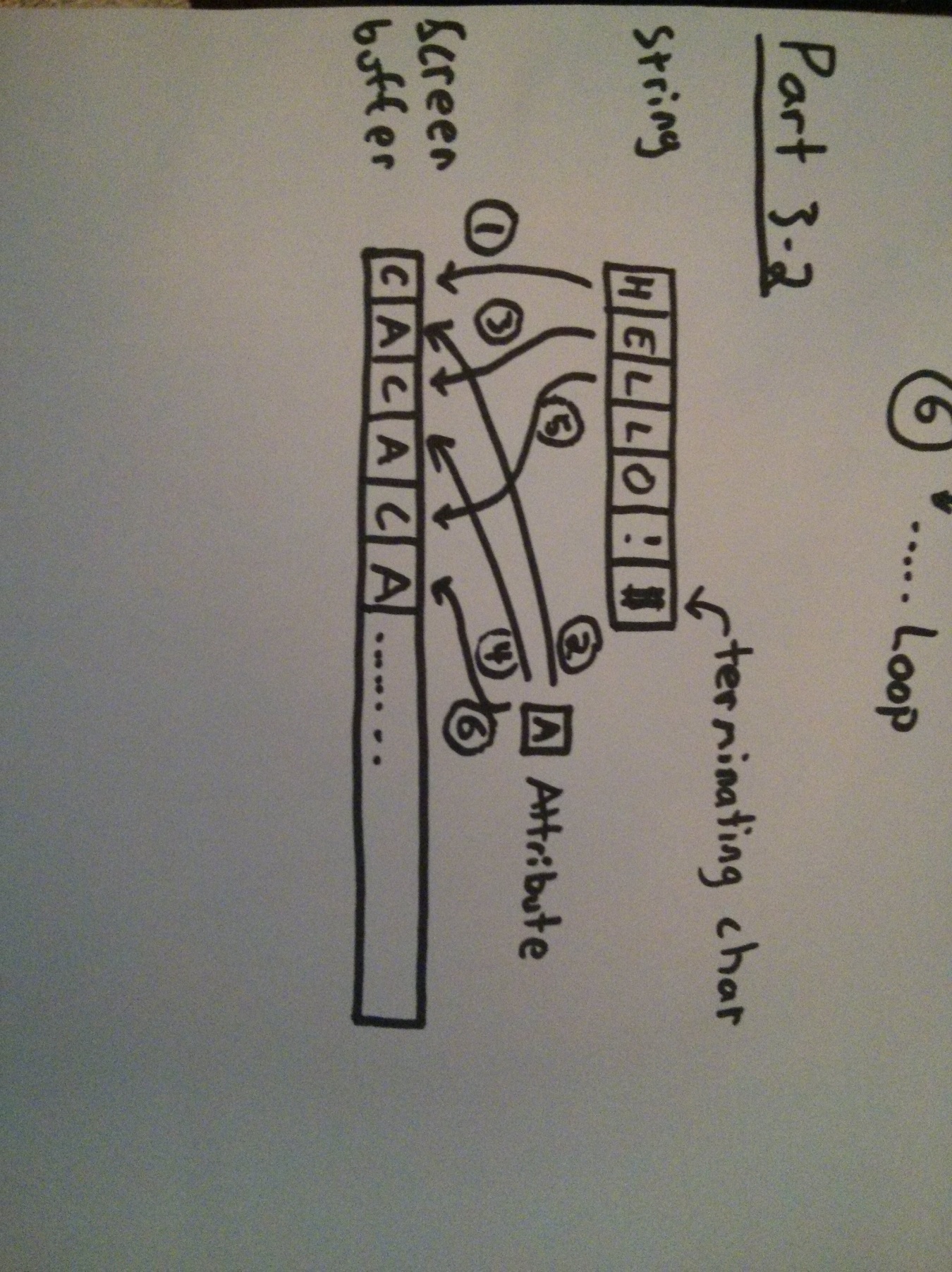
<http://www.petesqbsite.com/sections/tutorials/tuts/DEBUG.TXT>

<http://www.asciitable.com/>

<http://www.penguin.cz/~literakl/intel/intel.html>

# 3.0 – Diagrams

  
Diagram - Illustration of the general process used in part 2

  
Diagram - Illustration of the general process used in part 3-1  
  
Diagram - Illustration of the general process used in part 3-2

# 4.0 Implementation

## 4.1 Part 1 - IEEE floating point format

The data (value to be stored as a single or double precision float) was read using “scanf” as it is available in the “stdio.h” library which is available in both Linux and Windows. The data is stored in both a single and double precision float.

An unsigned char pointer is used to print out each byte of the stored data in both of the floats. A char pointer is used as it points to a single byte which is what we require. In order to assign the float address to the pointer we must cast the float to a char, demonstrated in line 19 of 8.1 Part 1 Code – “315P0AS.c”.

The data is then output to the screen by utilizing a for loop to iterate through every byte in the single or double float as demonstrated in line 20 of 8.1 Part 1 Code – “315P0AS.c”. The use of “limits.h” is used to determine the size of the data structure for this iteration.

Finally, as this system is little endian, both the single and double precision floats are printed out in “normal” hexadecimal order. From a comparison with an online calculator1, I can conclude that the system also conforms to the IEEE standard; online literature regarding the Visual Studio, and GCC compiler support this conclusion as well2.

1 - <http://www.h-schmidt.net/FloatConverter/IEEE754.html>

2 - <http://msdn.microsoft.com/en-us/library/e7s85ffb.aspx>

## 4.2 Part 2 – Capitalize and Reverse String

A buffer is set up by setting the register “BX1” at an empty space of memory and setting the value stored at [BX]2 equal to “0xFF3”, which corresponds to a maximum input of 255 characters. The data is then read into the input buffer using the MS-DOS system call “int 21/ 0x0A” (See lines 1-5 of 8.2 Part 2 Code – “part2\_debug.asm”). Lines 9-12 of 8.2 Part 2 Code – “part2\_debug.asm”) check to make sure the buffer isn’t empty.

Using the value stored at [BX+1], which is the number of bytes read, as an index from the start of the buffered string, BX+2, the position in memory of the last character can be found ([BX+1] + (BX+1)). The character being processed is then tested to see if it is lowercase, by checking that its ASCII value is greater than 0x60. If the value is lowercase it is uppercased by subtracting 0x20 from it. Finally it is output to the screen using the MS-DOS call “int 21/0x02”. This process repeats until all of the characters have been output to the screen and then the program exits. See Diagram 1 for a visual flow of this process and lines 14-31 of 8.2 Part 2 Code – “part2\_debug.asm”.

1 – A register defined by its label refers to the value stored in it.

2 – A register in “[]” –square brackets – refers to the value pointed to by the address in that register

3 – The 0x00 notation is used to refer to a hexadecimal number.

## 4.3.1 Part 3-1 – Randomize color of screen

First ah is checked and a jump is used to either move to the code that performs part 3-1 or part 3-2. See lines 1-4 of 8.3 Part 3 Code – “part3\_debug.asm”.

Once in the proper “system call”, BX is set to point to the video buffer (See lines 4&5 of 8.3 Part 3 Code – “part3\_debug.asm”). The screen color is pseudo-randomized by XORing the attribute and character values together and then propagating this value as a seed for the next set of XORs. See Diagram 2 for a visual flow and lines 10-12 of 8.3 Part 3 Code – “part3\_debug.asm”. BX is then incremented by 0x02 to skip over the next ASCII byte and to point at the next attribute byte.

The XORing process is repeated for every attribute byte on the screen using the loop conditions at lines 14-16 of 8.3 Part 3 Code – “part3\_debug.asm”.

The “system call” then exits.

## 4.3.2 Part 3-2 – Output String to any part of screen

After jumping to this “system call”, BX is preserved by pushing it onto the stack while the Direction Flow flag is set, and then restored (See lines 19-26 of 8.3 Part 3 Code – “part3\_debug.asm”). ES:DI and DS:SI are setup to point to the correct offset position in the video buffer and the input string respectively, remembering that we each position occupies two bytes (See lines 27-36 of 8.3 Part 3 Code – “part3\_debug.asm”).

The string is then output by moving the current character, as pointed to by DS:SI, to ES:DI which corresponds to its target position by using “movsb”. The next character of the string is preserved by moving it into a temporary register and pushing SI onto the stack so that we may utilize “movsb” to set the attribute byte. The value at [SI] is then replaced with the attribute data so that “movsb” sets the attribute for the current character (See lines 38-41 of 8.3 Part 3 Code – “part3\_debug.asm”). SI is then restored by popping it from the stack, and the next character is restored from its temporary register. See lines 42-45 of 8.3 Part 3 Code – “part3\_debug.asm”. Finally, the next character is checked for the termination character “$” and if found the program exits, otherwise it loops and outputs the next character (Lines 46-48 of8.3 Part 3 Code – “part3\_debug.asm”). See Diagram 3 for a visual flow of the data moving process from the string buffer to the screen buffer.

# 5.0 Results & Test Cases

Figures 1-4 below demonstrate the successful operation of all parts of the lab. Parts 2 & 3 also have “.com” files that can be executed in MS-DOS that show the proper output for a set of test cases. In addition, hex files of parts 2 & 3 have been included that are simply assembled versions of the “.asm” versions of the code that are able to be loaded quickly into DEBUG.exe by using “-n part2.hex” then “-l” then setting the registers as appropriate and if needed. This was done to improve the speed of testing by foregoing the need to retype the instructions every time by hand in DEBUG.exe.

Part 1 was tested by entering in different values and comparing the output against an online calculator1. The results were consistent across both Windows and Linux and matched the expected results.

Part 2 was fed a string of all lowercase letters, of mixed case letters, and a mixed string of letters and characters of both lower and upper case. As shown in Figure 1 - Part 1 Results, the various test cases all returned the expected results.

Part 3-1 was easily tested as its results are much more pass/fail than the other portions. Simply executing the program yields a pseudo-random color pattern on the screen as shown in Figure 3 - Part 3-1 Results \*\*New-line output after program exits and therefore not “dazzled.” The variety of the pattern is affected with different contents on screen and can be seen by outputting lots of text to the screen (ex. “-dir”) and executing the “system call”. The last line is not “randomized” as it is written to the screen buffer after the “system call” has finished which is expected behavior.

Part 3-2 was tested primarily using the code that was behind the “part3-2Demo.COM” as it sets the values of the string, the position, and the attribute color during executing meaning registers don’t need to be set every time which proved tedious and slow. This code only varied from the code at 8.3 Part 3 Code – “part3\_debug.asm” in that it set these values during execution. Figure 4 - Part 3-2 Results shows the result of this “system call” in Demo form which is identical to the behavior experienced when it is run from DEBUG with the exception of the above noted values being set during execution for simplicity; however, the results are consistently identical as expected. Of course, this was also compared against the output given while running the code at 8.3 Part 3 Code – “part3\_debug.asm” through Debug by setting the registers before execution and it yielded identical results. Note: The “$” character was chosen as the terminating character and needs to be appended to any string input.4.2 Part 2 – Capitalize and Reverse String

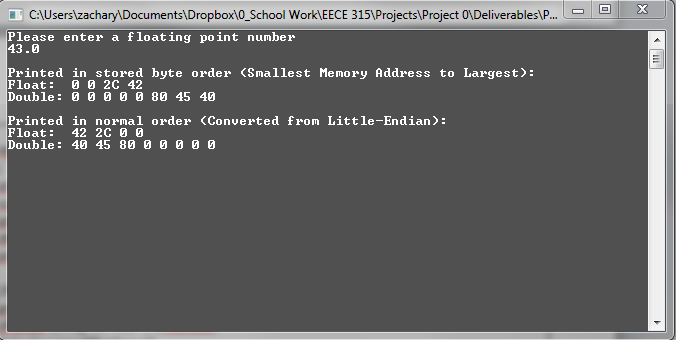


Figure - Part 1 Results

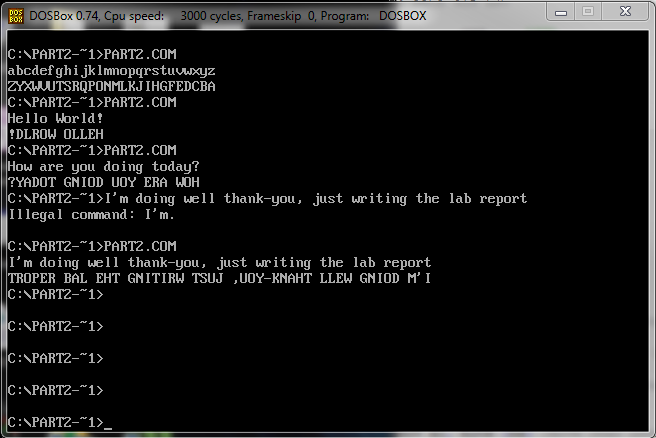


Figure - Part 2 Results

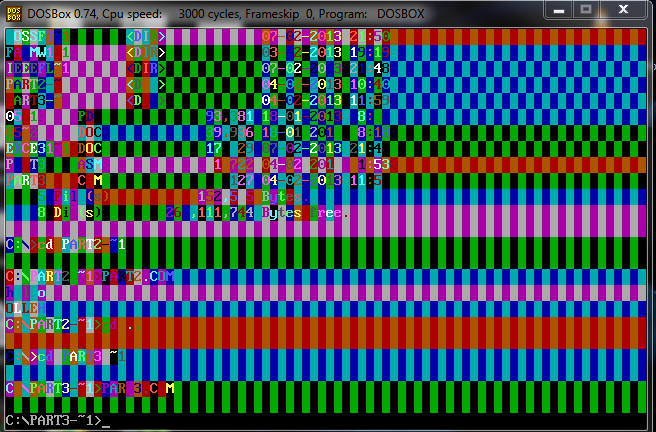


Figure - Part 3-1 Results \*\*New-line output after program exits and therefore not “dazzled.”

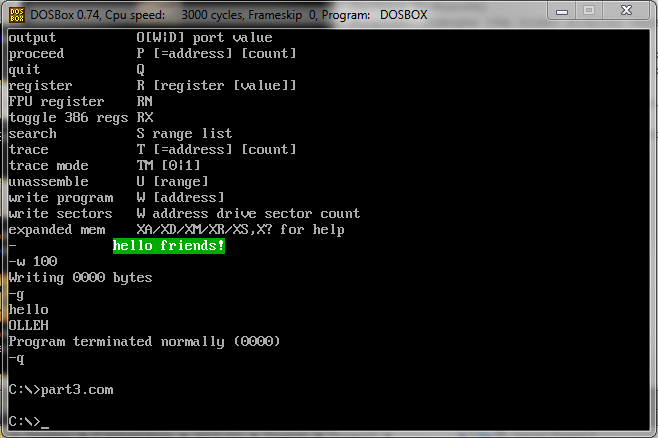


Figure - Part 3-2 Results

# 6.0 Testing

Testing was performed primarily by testing boundary values as well as an assortment of values within and outside of the input bounds to ensure the largest test coverage was achieved.

For Part 1 and Part 2: Repeated execution of the programs were performed with varying inputs and comparing the outputs against the expected results; which for part 1 were obtained through an IEEE 754 converter1, and in part 2 by inspecting the output strings to ensure all characters were capitalized and reversed in order.

For Part 2: Complete coverage of the case changing process was able to be tested by inputting the entire alphabet as shown in Figure 2 - Part 2 Results.

For Part 3-1: As the result is pass/fail in nature as to whether the colors are changed was easy to ascertain reliably through repeated testing. The pseudo-randomness was tested by having a variety of characters/colors on screen which was easily obtained by navigating the file tree or running other software that left colored text/backgrounds on screen. Using “Part3-2Demo.COM” greatly facilitated this process.

For Part 3-2: The robustness of the code was again tested by varying the input conditions repeatedly as well as testing the boundary conditions. By using the part3.hex file, I was able to test many cases quickly by simply adjusting the registers or string entered. Coverage was also improved by testing conditions such as when a string flows over its current line it increase the testing robustness.

1 - <http://www.h-schmidt.net/FloatConverter/IEEE754.html>

# 7.0 Comments & Conclusion:

All parts of this lab were successfully completed and the process was valuable in learning more about utilizing system calls, data structure storage, “interfacing” with hardware (Simulated CGA card), and of course problem solving and programming skills. I found the use of DEBUG.exe incredibly frustrating as it is not a true assembly programmer and that it no longer runs on the majority of modern hardware natively. By using an assembler (FASM) to target the x86 16-bit architecture instead of DEBUG.exe for my initial coding, I found I was able to learn significantly more, and quicker, about the system calls and assembly programming rather than stumble over the nuances of that particular program.

# 8.0 Appendix

## 8.1 Part 1 Code – “315P0AS.c”

|  |  |
| --- | --- |
|  | #include <stdio.h> |
|  | #include <limits.h> |
|  |  |
|  | float fNumber; |
|  | double dNumber; |
|  | unsigned char \*cPointer; |
|  | int x; |
|  |  |
|  | int main(int argc, char \*\*argv) |
|  | { |
|  | printf("Please enter a floating point number\n"); |
|  | scanf("%f",&fNumber); |
|  | getchar(); |
|  | dNumber=fNumber; |
|  | printf("\n"); |
|  |  |
|  | //Prints the contents of float |
|  | printf("Printed in stored byte order (Smallest Memory Address to Largest):\nFloat:\t"); |
|  | cPointer=(unsigned char\*) &fNumber; |
|  | for(x=0;x<sizeof(float);x++){ |
|  | printf("%X ", \*(cPointer+x)); |
|  | } |
|  | printf("\nDouble:\t"); |
|  | //Prints the contents of double |
|  | cPointer=(unsigned char\*) &dNumber; |
|  | for(x=0;x<sizeof(double);x++){ |
|  | printf("%X ", \*(cPointer+x)); |
|  | } |
|  | printf("\n\n"); |
|  |  |
|  | //Prints the contents of float |
|  | printf("Printed in normal order (Converted from Little-Endian):\nFloat:\t"); |
|  | cPointer=(unsigned char\*) &fNumber; |
|  | for(x=sizeof(float)-1;x>=0;x--){ |
|  | printf("%X ", \*(cPointer+x)); |
|  | } |
|  | printf("\nDouble:\t"); |
|  | //Prints the contents of double |
|  | cPointer=(unsigned char\*) &dNumber; |
|  | for(x=sizeof(double)-1;x>=0;x--){ |
|  | printf("%X ", \*(cPointer+x)); |
|  | } |
|  |  |
|  | getchar(); |
|  | return 0; |
|  | } |

## 8.2 Part 2 Code – “part2\_debug.asm”

|  |  |
| --- | --- |
|  | mov ah,0A |
|  | mov dx, 0200 |
|  | mov bx, dx |
|  | mov byte ptr [bx], ff |
|  | int 21 |
|  | mov ah, 02 |
|  | mov dl, 0A |
|  | int 21 |
|  | mov bx, 0201 |
|  | mov cl, [bx] |
|  | mov ax, 0000 |
|  | add al, cl |
|  | JZ 0146 |
|  | mov ax, 0201 |
|  | add al, [bx] |
|  | mov bx, ax |
|  | mov ah, 02 |
|  | mov al, [bx] |
|  | sub al, 60 |
|  | jbe 0133 |
|  | mov al, [bx] |
|  | sub al, 20 |
|  | jmp 0135 |
|  | mov al, [bx] |
|  | mov dl, al |
|  | int 21h |
|  | sub bx, 01 |
|  | sub cl, 01 |
|  | mov ax, 0000 |
|  | add al,cl |
|  | JNZ 0125 |
|  | mov ah, 4c |
|  | int 21 |

## 8.3 Part 3 Code – “part3\_debug.asm”

|  |  |
| --- | --- |
|  | mov al, 00 |
|  | sub ah, 01 |
|  | JZ 0109 |
|  | JMP 012a |
|  | mov ax, B800 |
|  | mov ds, ax |
|  | mov bx, 0000 |
|  | mov cl, [bx] |
|  | clc |
|  | xor cl, [bx] |
|  | xor cl, [bx+01] |
|  | mov [bx+1], cl |
|  | add bx, 02 |
|  | mov ax, bx |
|  | sub ax, 0FA0 |
|  | JB 0113 |
|  | mov ah, 4C |
|  | int 21 |
|  | push bx |
|  | mov bx, fbff |
|  | pushf |
|  | pop ax |
|  | and ax, bx |
|  | push ax |
|  | popf |
|  | pop bx |
|  | mov si, bx |
|  | mov ax, B800 |
|  | mov es, ax |
|  | mov ax, 00A0 |
|  | mul cl |
|  | mov bx, ax |
|  | mov ax, 0002 |
|  | mul ch |
|  | add ax, bx |
|  | mov di, ax |
|  | movsb |
|  | push si |
|  | mov al, [si] |
|  | mov [si], dl |
|  | movsb |
|  | pop si |
|  | mov [si], al |
|  | mov ah, 00 |
|  | mov al, [si] |
|  | sub ax, 24 |
|  | jz 0161 |
|  | jmp 14c |
|  | mov ah, 4C |
|  | int 21 |