ECE 441 Final Project Report

Spring 2016

By: Daniel H. Meves

Instructor: Dr. Jafar Saniie

TA: Boyang Wang

ECE 441-001

Due Date: 04-27-2016

Acknowledgment: I acknowledge all of the work (including figures, codes and writings) belongs to me.

Signature : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Table of Contents**

Abstract 3

Implementation of Commands 4

Quick Manual 8

Engineering and Design Challenges 11

Expansion of the Existing Monitor Program 12

Conclusion 14

References 15

**Abstract**

Resident monitor programs have been a part of computers since the early days of punch card computing and simple batch systems. They were simple versions of the modern operating system. The goal of this Design Project is to build up a Monitor program using the MC68000 assembly language. The monitor we designed is capable of performing basic debugging functions like memory display, memory sort, memory modify, block fill, block search, and block move. The monitor also can process exceptions through its exception handling routines.

**Implementation of Commands**

1. **Help**

The HELP command can be executed either with no arguments or with the argument of a command name. The MAN command is an alias of the HELP command, meant to provide familiarity to users of GNU/Linux & Unix man pages. The HELP command works by checking if an argument is provided. If there is no argument, the HELP command prints out all help messages using trap task #14 of TRAP #15. If a command name is provided as an argument to the HELP command, the jump tables are scanned for the appropriate command name and the program jumps to outputting the appropriate detailed man page for that command.

1. **Memory Display (MDSP)**

The MDSP command can be executed with one or two arguments. If only one argument is provided, the command will print memory data starting at the address specified by the 1st argument and ending after printing 16 bytes of data. If two arguments are provided, the command will print memory starting at the address specified by the 1st argument and ending at the address specified by the 2nd argument. Sane error checking is enforced by ensuring that the addresses provided are legitimate values.

1. **Sort Word (SORTW)**

The SORTW command must be executed with three arguments. The 1st argument specifies the beginning address in memory to be sorted. The 2nd argument specifies the ending address in memory to be sorted. The 3rd argument specifies the direction in which the sort will execute, ascending or descending. Sane error checking is enforced to ensure that legitimate hexadecimal addresses are provided. The algorithm used to sort the memory range by words is bubble sort. The implementation of this sorting algorithm was provided in laboratory 2. The only change made to the algorithm was the addition of the descending order code.

1. **Memory Modify (MM)**

The MM command must be executed with two arguments. The 1st argument specifies the location in memory to modify. The 2nd argument specifies the size of memory to display and edit: byte, word, or long. Three different handlers and print loops are coded for each of the different sizes. Sane error checking is enforced to ensure that a legitimate hexadecimal address is provided.

1. **Memory Set (MS)**

The MS command must be executed with two arguments. The 1st argument specifies the location in memory to be set. The 2nd argument specifies the data to be set into memory. The data can be either ASCII or hexadecimal and will be stored by executing a specific data store loop for each type of data. Sane error checking is enforced to differentiate between hexadecimal and ASCII data. If the data entered is not hexadecimal, it is always stored as ASCII, regardless of whether or not the string is comprehensible.

1. **Block Fill (BF)**

The BF command must be executed with three arguments. The 1st argument specifies the beginning address in memory to be filled. The 2nd argument specifies the ending address in memory to be filled. Both of these addresses must be even. The 3rd argument specifies a word data pattern to be filled in memory. Sane error checking is enforced to ensure that both addresses are even and legitimate values. The data is filled a word at a time into memory pointed to by the 1st argument. This process is looped until the 1st argument pointer is the same as the 2nd argument pointer.

1. **Block Move (BMOV)**

The BMOV command must be executed with three arguments. The 1st argument specifies the beginning address in memory to be moved. The 2nd argument specifies the ending address in memory to be moved. The 3rd argument specifies the location in memory to which this block will be moved. Sane error checking is enforced to ensure that all addresses are legitimate hexadecimal values. Data in memory will be moved byte by byte from the 1st argument pointer to the 3rd argument pointer. This move will loop until the 1st argument pointer is the same as the 2nd argument pointer.

1. **Block Test (BTST)**

The BTST command must be executed with two arguments. The 1st argument specifies the beginning address in memory to be tested. The 2nd argument specifies the ending address in memory to be tested. Sane error checking is enforced to ensure that both addresses are legitimate hexadecimal values. The test value is first moved into the memory block byte by byte until the pointer to the 1st argument is equal to the pointer to the 2nd argument. Then, the test value is compared to the same block of memory, byte by byte, to detect if any errors were made in storing the data. For every byte of memory that was successfully tested, a success message is outputted with the memory address. For every byte of memory with an error, an error message is outputted with the memory address.

1. **Block Search (BSCH)**

The BSCH command must be executed with three arguments. The 1st argument specifies the beginning address in which to search. The 2nd argument specifies the ending address in which to search. The 3rd argument specifies the string literal for which will be searched in memory. Sane error checking is enforced to ensure that both addresses are legitimate hexadecimal values. The ARGV subroutine has a quote check functionality that determines if a quote is open or closed. By using this functionality, the 3rd argument will be maintained as a string in memory even if spaces are present between words in the string. The ARGV function does not separate spaced words into arguments in the presence of an open quote. Only when an even number of quotes are seen (closed quote) will the ARGV function separate arguments. The BSCH command also checks if quotes are present in the string and appropriately replaces them with NULL characters before searching in memory for the string. This command is capable of finding a string anywhere within the bounds of the 1st and 2nd argument. The string does not have to be only at the beginning of the memory block.

1. **Go (GO)**

The GO command must be executed with one argument. The 1st argument specifies the location in memory to jump to in order to execute a user program. Sane error checking is enforced to ensure that this address is a legitimate hexadecimal value. The user program must end with the RTS command to successfully return to the resident monitor.

1. **Display Formatted Register (DF)**

The DF command must be executed with no arguments. Even if arguments are entered, the same functionality will occur. All registers will be outputted as well as the User and Supervisor stack pointers, status register, and the program counter. All D and A registers are retrieved from the global stack saved at the beginning of the MAIN program. The pointer to this global stack is stored in memory at SSPDATA. The stack pointers are output from the USPDATA and SSPDATA2 reserved memory locations. The status register is output from SRDATA and the program counter from PCDATA. These memory locations are initialized in the main program and can be updated as needed by subroutines such as the exception handlers. The exception handlers modify and restore the values of these memory locations.

1. **Exit**

The EXIT command must be executed with no arguments. The command works by calling trap task #9 of TRAP #15. This task number terminates the program by halting the simulator.

1. **Change/Modify Registers**

The modify registers command can be executed with no arguments or with an argument of the data to be inputted into the register. If no arguments are specified, the register is not updated with any values. If data is specified in the 1st argument, the register value is changed to that data, provided it is a legitimate hexadecimal value. The global state of the stack is saved at the beginning of the main program. The pointer to this global state is saved in memory. This memory location is accessed in each of the modify registers subroutines. The register values in this global stack are updated using address register indirect with displacement address mode of the MC68000.

1. **Clear**

The CLEAR command must be executed with no arguments. The command works by calling trap task #11 of TRAP #15 with D1.W set to $FF00. This program code clears all text on the output window/terminal screen.

1. **Command History (HISTORY)**

The HISTORY command can be executed either with no arguments or with an argument of number of commands to display. The command works by saving the input buffer to the history buffer in every iteration of the main program. The history buffer places a NULL character between each command. This allows for a print loop to be executed using trap task #14 of TRAP #15 in the command subroutine. The loop can be exited early by specifying an argument number of commands to display. The loop iterates over the history buffer the number of times specified in this argument.

**Quick Manual**

HELP: Displays this message

MDSP: Outputs Address and Memory Contents

MDSP <address1> <address2> eg: MDSP $908 $90A<CR>

SORTW: Sorts A Block of Memory

Default: Descending Order

A: Ascending Order'

D: Descending Order

SORTW <address1> <address2> [order] eg: SORTW $904 $90E A<CR>

MM: Modifies Data in Memory

Default: Displays One Byte

W: Displays One Word

L: Displays One Long Word

MM <address> [size] eg: MM $904 W<CR>

Exit MM command by typing "."<CR>

MS: Memory Set

MS <address> [data] eg: MS $904 $FFFF<CR>

BF: Fills A Block of Memory

BF <address1> <address2> <word> eg: BF $904 $908 475A<CR>

BMOV: Moves A Block of Memory to Another Area

BMOV <address1> <address2> <address3> eg: BMOV $908 $90B $909<CR>

BTST: Block Test

BTST <address1> <address2> eg: BTST $900 $90A<CR>

BSCH: Searches A Literal String In The Memory

BSCH <address1> <address2> "literal string" eg: BSCH $900 $910 "MATCH"<CR>

GO: Starts Execution from Given Address

GO <address> eg: GO $900<CR>

DF: Display Formatted Registers eg: DF<CR>

EXIT: Exit the monitor program eg: EXIT<CR>

Modify Registers: Modify the contents of each register individually

.D[0-7] [data] / .A[0-7] [data] / .SR [data] eg: .D4 $ABCD<CR>

HISTORY: Display Command History

N: Number of commands to display

HISTORY with no arguments prints all history

HISTORY [N] eg: HISTORY 10<CR>

CLEAR: Clear text from terminal eg: CLEAR<CR>

**Engineering and Design Challenges**

Writing an assembly program of this magnitude was quite challenging for a novice programmer such as myself. One major challenge was code modularization. In an effort to reuse as much code as possible, I tried to encapsulate frequently used blocks of code into functions/subroutines. This was easy for obvious functions such as ASCII to Hexadecimal and Hexadecimal to ASCII conversion. However, I failed to find an efficient way to print the help messages for each command when only the command name is typed. I also had issues in developing a uniform way of pointing address registers to command arguments. These two examples are cases in which I needed to custom tailor the code for each situation/command subroutine.

Another challenge was the length of the entire program and its effect on the behavior of the Easy68k simulator/assembler software. As my monitor program grew in length, I noticed slow down and lag in both the text editor/assembler and the simulator program. One way in which I counteracted this negative effect was to divide my code into separate files based on subroutine. I then made a single header file with all the INCLUDE statements for the individual files. It is from this file that you must execute the program.

**Expansion of the Existing Monitor Program**

If given more time to work on the project, I would like to add additional commands to the Monitor program. I originally wanted to incorporate the up and down arrow keys into the command history command. This would have been implemented using trap task 19 of the trap 15 I/O commands. The problem I ran into when implementing this feature was the inability to accept input on the command line in the main program while also checking for up and down arrow key presses. Given the single threaded nature of the 68k simulator, only one instruction can be executed at a time. Therefore, it may not be feasible to allow scrolling through the command history with the arrow keys unless we were developing for a multi core microprocessor (the M68000 is not one unfortunately).  
 It would also be interesting to rewrite the monitor program to use TRAP #14 on the SANPER-1 ELU hardware. This would allow us to implement commands that communicate with external devices via the PIA and ACIA interfaces.

Because resident monitor programs are primitive versions of modern operating systems, developing a full-fledged OS for an embedded system using the MC68000 would mean building up a program similar to the one we built in this design project. An example of a more advanced implementation of this monitor program that would help in building an OS is the addition of networking. We may be able to achieve a primitive form of networking using the ACIA interface. We could also design a new board to insert into the SANPER unit that features a RJ45 port. This would also require that we write a driver/handler for this networking card.

In order to save time in the development of an embedded system operating system, it might be beneficial to rewrite this monitor program and associated drivers in a higher level language such as C. This comes with added benefit of code portability. However, some features such as our custom exception vector handlers may still have to be written in assembly. It may not be possible to modify the exception vector table in a higher level language such as C.

**Conclusion**

Completing this design project has provided me with confidence in my understanding of assembly language programming and the Motorola MC68000 Microprocessor. Although this project took many hours of labor to complete, I was able to successfully debug my program. All known issues were resolved by the time of presentation.

Due to the limited use of this microprocessor, I do not think this monitor program has much practical use outside this course. However, the concepts and methods learned in the design of a resident monitor program can be used to design software for other more modern microprocessors. Overall, I think that this project was beneficial to my learning and improvement.

**References**

[1] Motorola MC 68000 Microprocessor Family – Second Edition

[2] Motorola M68000 Programmer’s Reference Manual

[3] Motorola M68000 8-/16-/32-Bit Microprocessors User’s Manual – Ninth Edition

[4] Microprocessor Systems Design – 68000 Hardware, Software, and Interfacing – Third Edition

[5] “The command-line history.” *The Linux Documentation Project*. The Linux Documentation

Project, 27 April 2016. Web. 27 April 2016. <http://www.tldp.org/LDP/GNU-Linux-Tools-Summary/html/x1712.htm>.