ECE 441
Microprocessors

Instructor: Dr. Jafar Saniie
Teaching Assistant: Guojun Yang

Final Project Report:
MONITOR PROJECT
04/26/17

By: Javier Sorribes

Acknowledgment: I acknowledge all of the work including figures and codes are belongs to me and/or persons who are referenced.

Signature : ______________________________________

Table of Contents

1-) Abstract

The monitor program allows a user to enter commands and interact with a Motorola MC68000 processor and its memory.

In this report, all available commands are described, along with their implementations. Their corresponding algorithms, flowcharts and assembly codes are provided.

A similar description of the exception handling subroutines follows.

In addition, a quick user manual with the command usage can be found.

Furthermore, a discussion about challenges and uses of this project is proposed. This section is continued with feature suggestions for newer, more advanced versions of this program. Some conclusions about the project itself are also given.

Finally, external references and an Appendix with all of the code in the monitor program are provided at the end of the report.

By reading this report, the user will understand the usage and implementation of the monitor program, as well as the design and production process.

2-) Monitor Program

This program allows the user to enter an executable command into the console, sometimes providing the appropriate arguments. Then, the command is run, the output (if any) displayed. Finally, the prompt will be redisplayed and the process will start over. The user may run the 'EXIT' command to terminate the program. The following flowchart represents this process:

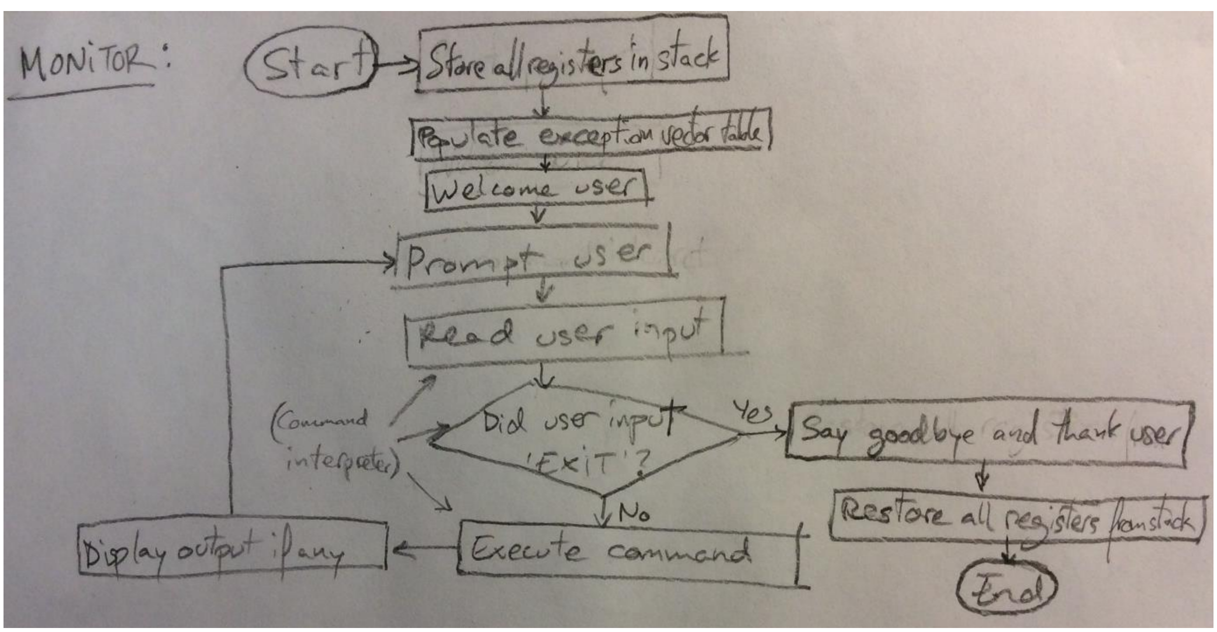

Figure 2.1. Monitor Program Flowchart

Descriptions of all the available commands, also named debugger commands for their hardware debugging capabilities, are outlined in the following sections.

In addition, this program accounts for asynchronous exceptions, providing exception handling routines. Their descriptions can also be found in the sections after those dedicated to the debugger commands.

Finally, note that many of the subroutines outlined in this report use other helper subroutines. These are not explicitly explained in this report, but refer to the Appendix for the code in these subroutines, which are simple enough to be understood from the assembly code and comments.

2.1-) Command Interpreter

The command interpreter compares the first word of the input against a table with all command names. These command names are preceded with a digit determining their length, which is used by the algorithm to know how to advance to the next row, or name. Each command name is also followed by either a null or space character, depending on whether the command takes arguments or not.

Once the command interpreter finds the command name in the table, it uses the offset within that table to access the correct memory location of the executable in a command location table. If the name is not found, then an invalid message is displayed and the program prompts again.

Note that the command interpreter does not parse the arguments of the command, but rather leaves that task to each command. This design decision was taken because each command may require a variable number of commands in different formats.

2.1.1-) Algorithm and Flowchart

An algorithm of the design and its flowchart are displayed below:

```
COMMAND INTERPRETER
While input != 'EXIT'
    Print prompt
    Read input into the stack
    counter = 0
    row ← first row in command names table     // row is name with length preceding
    While row < last row in table
        counter2 ← length of name from row
        While counter2 > 0
            If next byte of input == next byte in row       // keep comparing
                counter2 = counter2 – 1
            Else            // name is different from input
                counter = counter + 1
                row ← next row
                Break while loop
```

End while
If counter > # command names // name not in table
Print invalid message
Break while loop
Else if counter = 0 // name was found
Execute command at offset counter from command addresses table
End while
End while
Finish // finish

Figure 2.2. Command Interpreter Algorithm

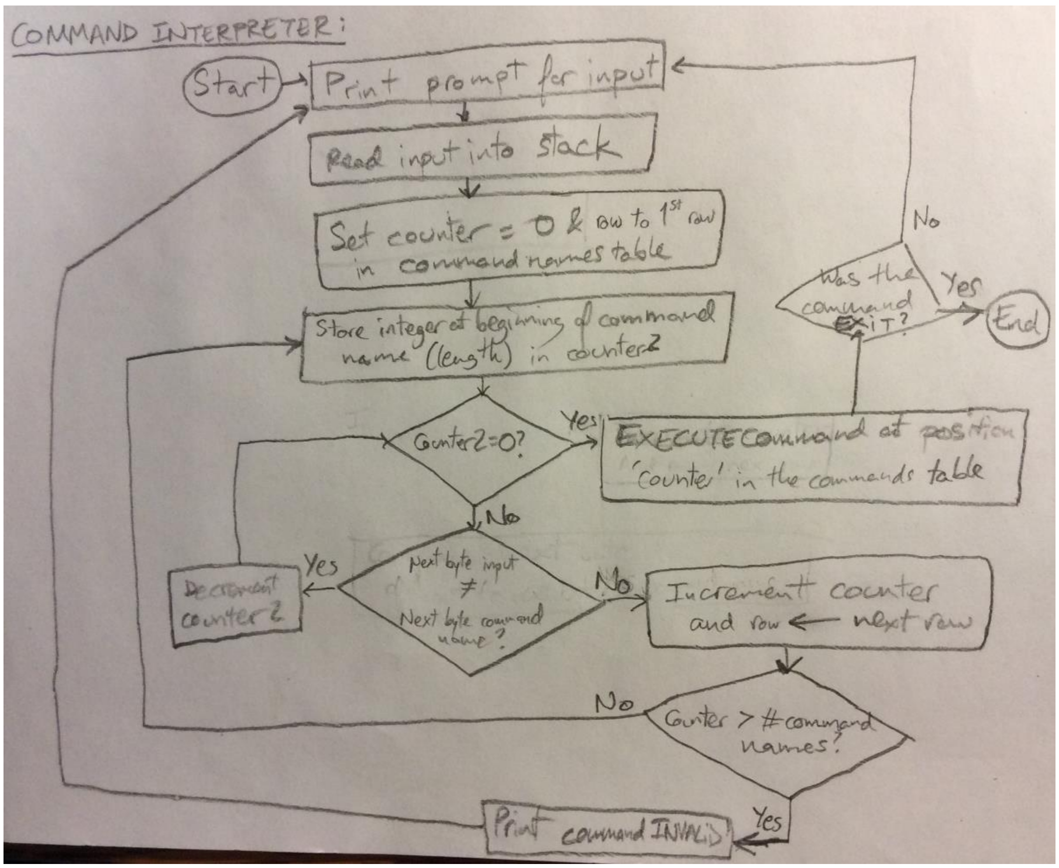

Figure 2.2. Command Interpreter Flowchart

2.1.2-) Command Interpreter Assembly Code

```
*** MAIN: Prompt, execute and repeat ***
        LEA     WELCOME,A1
        MOVE.B  #13,D0
        TRAP    #15      ; display welcome message
        SUBA.L  #MAX_IN_LEN,A7  ; open space in stack for input (do only once)
*** COMMAND INTERPRETER ***
PROMPT  LEA     PROMPT_STR,A1
        MOVE.B  #14,D0
        TRAP    #15      ; print out prompt
        MOVEA.L A7,A1    ; input will go in stack
        MOVE.B  #2,D0
        TRAP    #15      ; read user input, length stored in D1

        LEA     COM_TABL,A4 ; beginning of command table
        LEA     COM_ADDR,A5 ; end of command table
        CLR.L   D3       ; will be the count of where the command is
SEARCH  CLR.L   D2
        MOVE.B  (A4)+,D2    ; length of next command string
        SUBI.B  #$30,D2 ; convert ascii num to hex
        MOVEA.L A1,A6    ; pointer to input string
CMP_B   CMPM.B  (A4)+,(A6)+ ; compare byte to byte with command names
        DBNE    D2,CMP_B    ; keep comparing characters until length is over
        TST.W   D2
        BLT     EXEC     ; loop was exhausted and all chars were equal
        ADDA.L  D2,A4    ; go to end of command
        ADDQ.L  #2,D3    ; else, increment offset by word size
        CMPA.L  A4,A5 ; end of COM_TABL
        BGE     SEARCH   ; keep on searching

        BSR     INVALID ; print invalid command message
        BRA     PROMPT ; prompt again

EXEC    ADDA.L  D3,A5    ; add offset to COM_ADDR start
        MOVEA.L #0,A3    ; clear A3, used for subroutine call
        MOVEA.W (A5),A3 ; move that command's address to register
        JSR     (A3)     ; jump to that command's subroutine (below)

        BRA     PROMPT   ; prompt again
```

Figure 2.3. Main & Command Interpreter 68000 Assembly Code

2.2-) Debugger Commands

All debugger command subroutines store all used registers in the stack at the beginning and restore them at the end to ensure that nothing is overwritten. They each parse the arguments passed if anything, and display an invalid message if the usage is wrong. Then, they proceed to execute the corresponding algorithm and display any relevant output. Finally, they return to the main subroutine.

2.2.1-) Debugger Command #1: HELP

Displays the commands' descriptions and usage. Prints the message in two parts to avoid not showing a part of it if it is too long.

2.2.1.1-) Debugger Command #1 Algorithm and Flowchart

HELP
Print first part of message
While no input // wait
End while
Print second part of help message
Finish

Figure 2.4. Debugger Command # 1 Algorithm

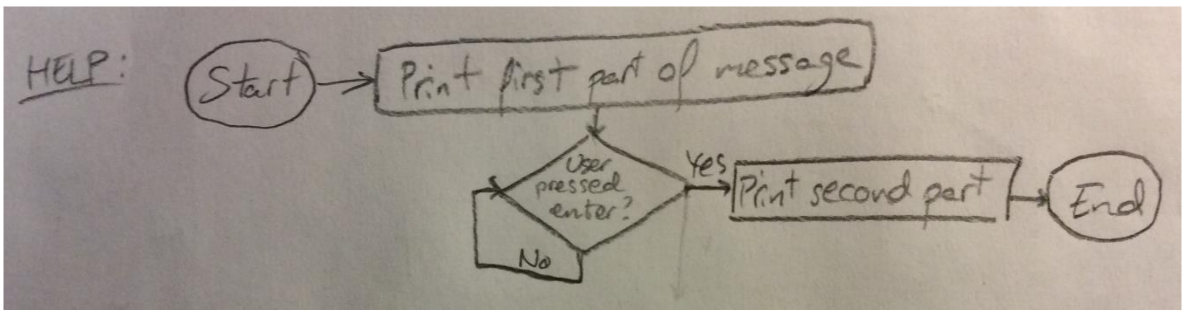

Figure 2.5. Debugger Command # 1 Flowchart

2.2.1.2-) Debugger Command #1 Assembly Code

```
* HELP -- displays help message
HELP    MOVEM.L D0-D1/A1,-(A7) ; store used registers in stack
        LEA     HELP_MSG,A1
        MOVE.B  #13,D0
        TRAP    #15     ; print first part of the help message
        MOVE.B  #5,D0
        TRAP    #15     ; wait for the user to enter a character
        LEA     HELP_MSG2,A1
        MOVE.B  #13,D0
        TRAP    #15     ; print second half of the message
        MOVEM.L (A7)+,D0-D1/A1 ; restore registers from stack
        RTS
```

Figure 2.6. Debugger Command #1 Assembly Code

2.2.2-) Debugger Command #2: MDSP

Displays contents of memory between address1 (inclusive) and address2 (exclusive) word by word.

2.2.2.1-) Debugger Command #2 Algorithm and Flowchart

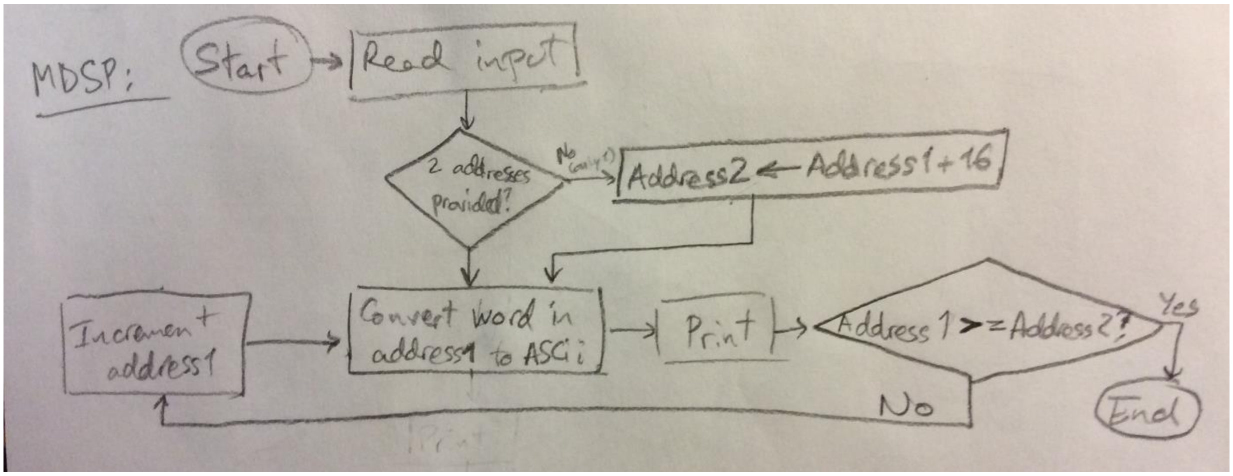

Figure 2.7. Debugger Command #2 Flowchart

2.2.2.2-) Debugger Command #2 Assembly Code

```
* MDSP -- displays memory block
MDSP     MOVEM.L D0-D4/A1-A4,-(A7)
         MOVE.B  (A6)+,D1     ; first '$'
         CMPI.B  #$24,D1 ; is it '$'?
         BNE     MDSPINV ; wrong command usage
         BSR     MEM2HEX ; D1 has 1st address in hex
         MOVEA.L D1,A2    ;store in A2
         MOVE.B  (A6)+,D1     ; space in between addresses
         TST.B   D1   ; if null, no 2nd address, so address2 = address1 + 16
         BNE     MDSPADDR2
         MOVEA.L A2,A3
         ADDA.L  #16,A3   ; A3 = A2 +16
         BRA     MDSPLOOP
MDSPADDR2    MOVE.B  (A6)+,D1     ; second '$'
         CMPI.B  #$24,D1
         BNE     MDSPINV
         BSR     MEM2HEX ; D1 has 2nd address in hex
         MOVEA.L D1,A3
MDSPLOOP     MOVEA.L A7,A1
         SUBA.L  #$40,A1 ; move A1 far from A7 to avoid collision in subroutines
         MOVE.B  #$00,-(A1) ; null terminator
         MOVE.B  #$20,-(A1)  ; space
         MOVE.B  #$3E,-(A1)  ; '>' for nicer output
         MOVE.L  A2,D1    ; memory address into D1
         BSR     HEX2MEM ; puts digits of D1 into -X(A1) in ascii (no trailing zeros)
         MOVE.B  #$24,-(A1)  ; '$' for nicer output
```

```
        MOVE.B  #14,D0
        TRAP    #15     ; print current memory address
        MOVE.B  #$00,-(A1)  ; null terminator
        MOVE.L  (A2)+,D1    ; memory value into D1
        BSR     HEX2MEM ; puts digits of D1 into -X(A1) in ascii (no trailing zeros)
        MOVE.B  #13,D0
        TRAP    #15     ; print
        CMPA.L  A2,A3
        BGT     MDSPLOOP
        BRA     MDSPDONE
MDSPINV BSR     INVALID ; print invalid command message
MDSPDONE    MOVEM.L (A7)+,D0-D4/A1-A4
        RTS
```

Figure 2.8. Debugger Command #2 Assembly Code

2.2.3-) Debugger Command #3: SORTW

Sorts a block of memory in between addresses 1 and 2 (inclusive) in either ascending or descending order. The command should be called in the form “SORTW <address1> <address2> A|D”, where A refers to ascending and D to descending (default).

The size of each number within the memory specified is expected to be word, and the type unsigned.

2.2.3.1-) Debugger Command #3 Algorithm and Flowchart

The algorithm for sorting is based on Bubble Sort, a method to “bubble up” items to their correct locations. By comparing numbers to the adjacent ones, we can decide whether to swap these or continue. Please refer to Lab Manual 2, Procedure 2.5 for more details.

In addition, a small check was implemented to be able to do either ascending or descending order as requested by the user.

```
SORTW                               // first line
Parse input to get 'start', 'end' and 'type' (A or D)
While start < end                   // start will serve as an incrementing pointer
    If start < start+1 and type = A // using start as an address pointer
        Swap start with start+1         // so start+1 is the item after start
        Reset start to original value (start over)
    Else if start > start+1 and type = D
        Swap start with start+1
        Reset start to original value
    Else                            // order is fine, move on to next
        start = start + 1
    End if
End while
Finish                              // finish
```

Figure 2.9. Debugger Command #3 Algorithm

The following flowchart is an abstraction of the algorithm described above:

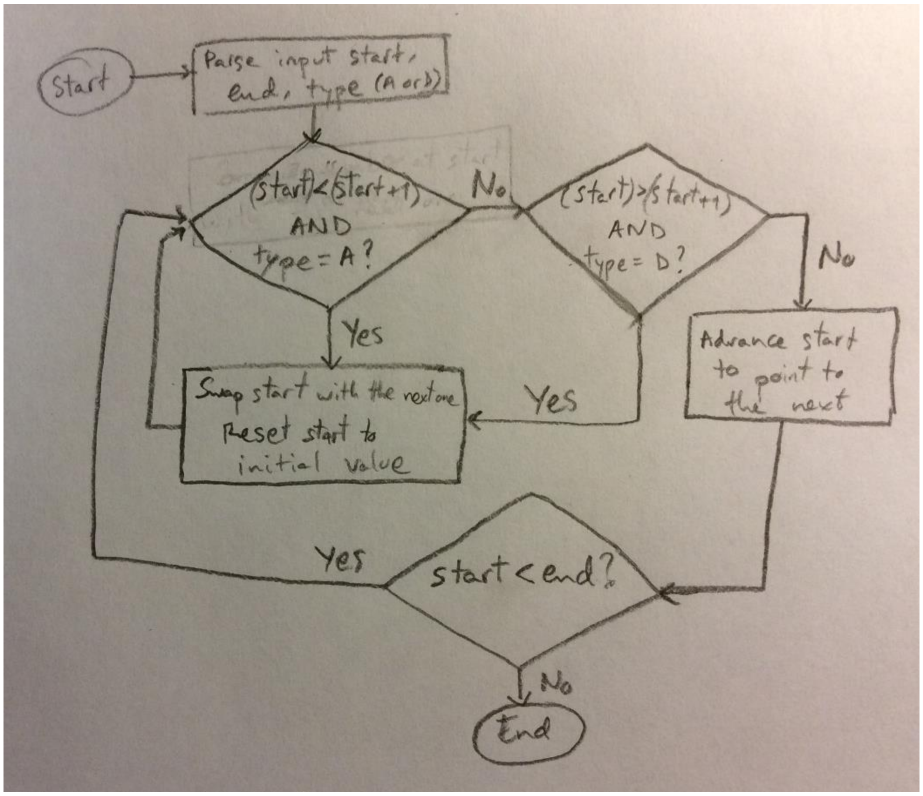

Figure 2.10. Debugger Command #3 Flowchart

2.2.3.2-) Debugger Command #3 Assembly Code

```
* SORTW -- implements bubble sort (unsigned numbers)
SORTW   MOVEM.L D0-D4/A1-A4,-(A7)
        MOVE.B  (A6)+,D1     ; first '$'
        CMPI.B  #$24,D1      ; is it '$'?
        BNE     SORTWINV     ; wrong command usage
        BSR     MEM2HEX      ; D1 has 1st address in hex
        MOVEA.L D1,A2        ; store in A2
        MOVE.B  (A6)+,D1     ; space in between addresses
        CMPI.B  #$20,D1      ; is it ' '?
        BNE     SORTWINV     ; wrong command usage
        MOVE.B  (A6)+,D1     ; second '$'
        CMPI.B  #$24,D1      ; is it '$'?
        BNE     SORTWINV     ; wrong command usage
        BSR     MEM2HEX      ; D1 has now the 2nd address
        MOVEA.L D1,A3        ; store in A3
        MOVE.B  (A6)+,D1     ; space
        CMPI.B  #$00,D1      ; is it NULL?
        BEQ     SORTWDEF     ; use default: descending (D1=0)
        CMPI.B  #$20,D1      ; or is it ' '?
        BNE     SORTWINV     ; wrong command usage
        MOVE.B  (A6)+,D1     ; char either 'A' or 'D'
        CMPI.B  #$41,D1      ; is it 'A'?
```

```
        BEQ      SORTWLOOP    ; if so, D1 marks ascending
        CMPI.B   #$44,D1      ; else, is it 'D'?
        BNE      SORTWINV     ; if it isn't, input was invalid
SORTWDEF    CLR.L    D1           ; if it is, D1=0 marks descending
SORTWLOOP   MOVEA.L A2,A4     ; first address copied into A4
SORTWCMP    TST.B    D1       ; tells us whether ascending or descending
        BEQ      SORTWD   ; do descending
SORTWA  CMP.W    (A4)+,(A4)+ ; compare next two numbers
        BCS      SORTWSWAP    ; swap if not in ascending order (if 1st>2nd)
        BRA      SORTWNEXT    ; otherwise, move on
SORTWD  CMP.W    (A4)+,(A4)+ ; compare next two numbers
        BHI      SORTWSWAP    ; swap if not in descending order (if 2nd>1st)
SORTWNEXT   SUBQ.L   #2,A4    ; look back at previous number
        CMP.L    A4,A3
        BNE      SORTWCMP     ; keep comparing if not at end yet (A3 inclusive)
        BRA      SORTWDONE    ; else, done
SORTWSWAP MOVE.L   -(A4),D4    ; move both words to register
        SWAP.W   D4   ; swap the two words
        MOVE.L   D4,(A4) ; write them back
        BRA      SORTWLOOP    ; loop again from start
SORTWINV    BSR INVALID
SORTWDONE   MOVEM.L (A7)+,D0-D4/A1-A4
        RTS
```

Figure 2.11. Debugger Command #3 Assembly Code

2.2.4-) Debugger Command #4: MM

Displays a byte, word or long in memory and allows the user to input a new value in hex. Starts at the address provided and goes on until the user inputs a period '.'.

2.2.4.1-) Debugger Command #4 Algorithm and Flowchart

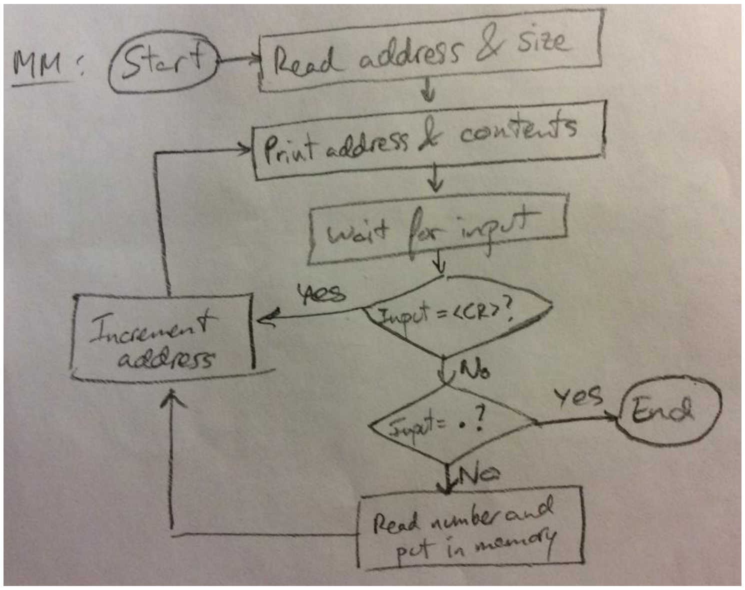

Figure 2.12. Debugger Command #4 Flowchart

2.2.4.2-) Debugger Command #4 Assembly Code

```
* MM -- modifies data in memory. Size can be B, W or L
MM      MOVEM.L D0-D1/A0-A1,-(A7)
        MOVEA.L A6,A1    ; A1 used for I/O later
        MOVE.B  (A6)+,D1     ; '$'
        CMPI.B  #$24,D1 ; is it '$'?
        BNE     INVALID ; wrong command usage
        BSR     MEM2HEX ; D1 has address in hex
        MOVEA.L D1,A0    ;store in A0
        MOVE.B  (A6)+,D1     ; ' ' before option
        CMPI.B  #0,D1        ; is it null?
        BEQ     MMBYTE  ; use default: byte
        CMPI.B  #$20,D1 ; is it ' '?
        BNE     INVALID ; wrong command usage
        MOVE.B  (A6)+,D1     ; the option
        CMPI.B  #'B',D1
        BEQ     MMBYTE
        CMPI.B  #'W',D1
        BEQ     MMWORD
        CMPI.B  #'L',D1
        BEQ     MMLONG
        BRA     MMINV   ; wrong option
MMBYTE  ADDA.L  #14,A1  ; output will be 13 chars long + null
        MOVE.B  #0,-(A1)     ; null terminator
        MOVE.B  #'?',-(A1)   ; nicer output
        CLR.L   D1
        MOVE.B  (A0),D1      ; content of memory to D1
        BSR     HEX2MEM      ; writes memory content to -8(A1)
        ADDA.L  #6,A1        ; we only want 2 chars, not 8
        MOVE.B  #$9,-(A1)    ; a tabspace
        MOVE.L  A0,D1        ; memory address
        BSR     HEX2MEM      ; memory address to -8(A1)
        MOVE.B  #'$',-(A1)   ; nicer output
        MOVE.B  #14,D0
        TRAP    #15          ; print
        MOVE.B  #2,D0
        TRAP    #15          ; read new value, if any
        CMPI.B  #0,(A1)
        BNE     MMBNEXT      ; skip memory address?
        ADDA.L  #1,A0        ; if yes, increment A0
        BRA     MMBYTE       ; ...and loop
MMBNEXT CMPI.B  #'.',(A1)    ; else, check if done (entered '.')
        BEQ     MMDONE
        MOVEA.L A1,A6        ; new value to write in!
        BSR     MEM2HEX      ; store input value from A6 in D1
        MOVE.B  D1,(A0)+     ; put it in address location
        BRA     MMBYTE       ; and loop!
MMWORD  ADDA.L  #16,A1  ; output will be 15 chars long + null
        MOVE.B  #0,-(A1)
        MOVE.B  #'?',-(A1)
```

```
        CLR.L   D1
        MOVE.W  (A0),D1
        BSR     HEX2MEM     ; writes memory content to -8(A1)
        ADDA.L  #4,A1       ; we only want 4 chars, not 8
        MOVE.B  #$9,-(A1)   ; a tabspace
        MOVE.L  A0,D1
        BSR     HEX2MEM     ; memory address to -8(A1)
        MOVE.B  #'$',-(A1)
        MOVE.B  #14,D0
        TRAP    #15         ; print
        MOVE.B  #2,D0
        TRAP    #15         ; read new value, if any
        CMPI.B  #0,(A1)
        BNE     MMWNEXT     ; skip memory address?
        ADDA.L  #2,A0       ; if yes, increment A0
        BRA     MMWORD      ; ...and loop
MMWNEXT CMPI.B  #'.',(A1)   ; else, check if done (entered '.')
        BEQ     MMDONE
        MOVEA.L A1,A6       ; new value to write in!
        BSR     MEM2HEX     ; store input value from A6 in D1
        MOVE.W  D1,(A0)+    ; put it in address location
        BRA     MMWORD      ; and loop!
MMLONG  ADDA.L  #20,A1  ; output will be 19 chars long + null
        MOVE.B  #0,-(A1)
        MOVE.B  #'?',-(A1)
        CLR.L   D1
        MOVE.L  (A0),D1
        BSR     HEX2MEM     ; writes memory content to -8(A1)
        MOVE.B  #$9,-(A1)   ; a tabspace
        MOVE.L  A0,D1
        BSR     HEX2MEM     ; memory address to -8(A1)
        MOVE.B  #'$',-(A1)
        MOVE.B  #14,D0
        TRAP    #15         ; print
        MOVE.B  #2,D0
        TRAP    #15         ; read new value, if any
        CMPI.B  #0,(A1)
        BNE     MMLNEXT     ; skip memory address?
        ADDA.L  #4,A0       ; if yes, increment A0
        BRA     MMLONG      ; ...and loop
MMLNEXT CMPI.B  #'.',(A1)   ; else, check if done (entered '.')
        BEQ     MMDONE
        MOVEA.L A1,A6       ; new value to write in!
        BSR     MEM2HEX     ; store input value from A6 in D1
        MOVE.L  D1,(A0)+    ; put it in address location
        BRA     MMLONG      ; and loop!
MMINV   BSR     INVALID
MMDONE  MOVEM.L (A7)+,D0-D1/A0-A1
        RTS
```

Figure 2.13. Debugger Command #4 Assembly Code

2.2.5-) Debugger Command #5: MS

Reads in an ASCII or hex value and places it in memory at the address specified.

2.2.5.1-) Debugger Command #5 Algorithm and Flowchart

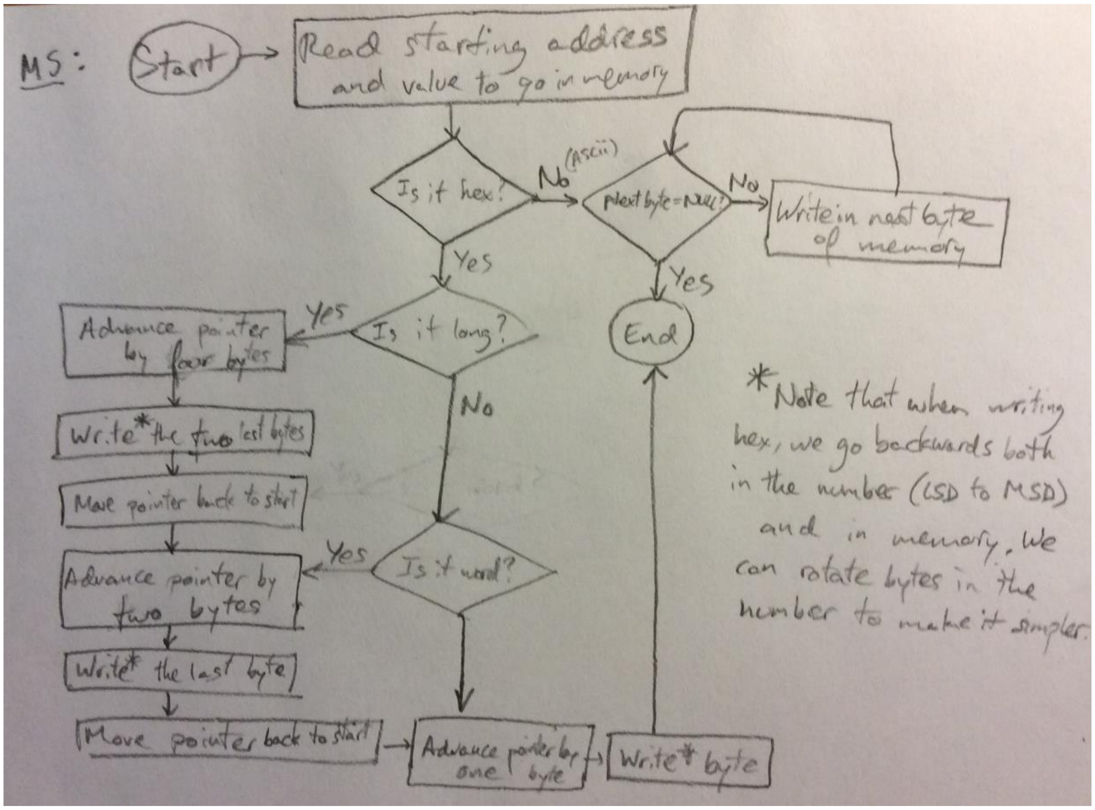

Figure 2.14. Debugger Command #5 Flowchart

2.2.5.2-) Debugger Command #5 Assembly Code

```
* MS -- store ascii (including null terminator) or hex in memory
MS      MOVEM.L D1/A1,-(A7)
        MOVE.B  (A6)+,D1      ; first '$'
        CMPI.B  #$24,D1       ; is it '$'?
        BNE     MSINV    ; wrong command usage
        BSR     MEM2HEX       ; D1 has 1st address in hex
        MOVEA.L D1,A1         ; store in A1
        MOVE.B  (A6)+,D1
        CMPI.B  #$20,D1       ; is it ' '?
        BNE     MSINV   ; wrong command usage
        MOVE.B  (A6)+,D1
        CMPI.B  #$24,D1       ; '$'?
        BEQ     MSHEX
        SUBA.L  #1,A6   ; have to put A6 back at start of ascii
MSASCII MOVE.B  (A6),(A1)+  ; put that char in (A1) and increment A1
        CMPI.B  #0,(A6)+      ; check if end and increment A6 to match A1
        BEQ     MSDONE  ; end of string
```

```
        BRA     MSASCII ; repeat
MSHEX   BSR     MEM2HEX ; hex number stored in D1
        CMPI.L  #$FF,D1 ; see size of number
        BLE     MSBYTE
        CMPI.L  #$FFFF,D1
        BLE     MSWORD
MSLONG  ADDA.L  #4,A1   ; move A1 to end of long word
        MOVE.B  D1,-(A1)    ; have to copy 4 bytes
        ROR.L   #8,D1       ; first one was copied, so look at next byte
        MOVE.B  D1,-(A1)    ; copy second byte
        ROR.L   #8,D1
        SUBA.L  #2,A1   ; done to counteract the next action
MSWORD  ADDA.L  #2,A1   ; move A1 to end of word
        MOVE.B  D1,-(A1)    ; will copy 2 bytes
        ROR.L   #8,D1   ; look at second one
        SUBA.L  #1,A1   ; to counteract the fact that MSBYTE doesn't predecrement
MSBYTE  MOVE.B  D1,(A1) ; copy one byte
        BRA     MSDONE
MSINV   BSR     INVALID
MSDONE  MOVEM.L (A7)+,D1/A1
        RTS
```

Figure 2.15. Debugger Command #5 Assembly Code

2.2.6-) Debugger Command #6: BF

It is similar to 2.2.1

2.2.6.1-) Debugger Command #6 Algorithm and Flowchart

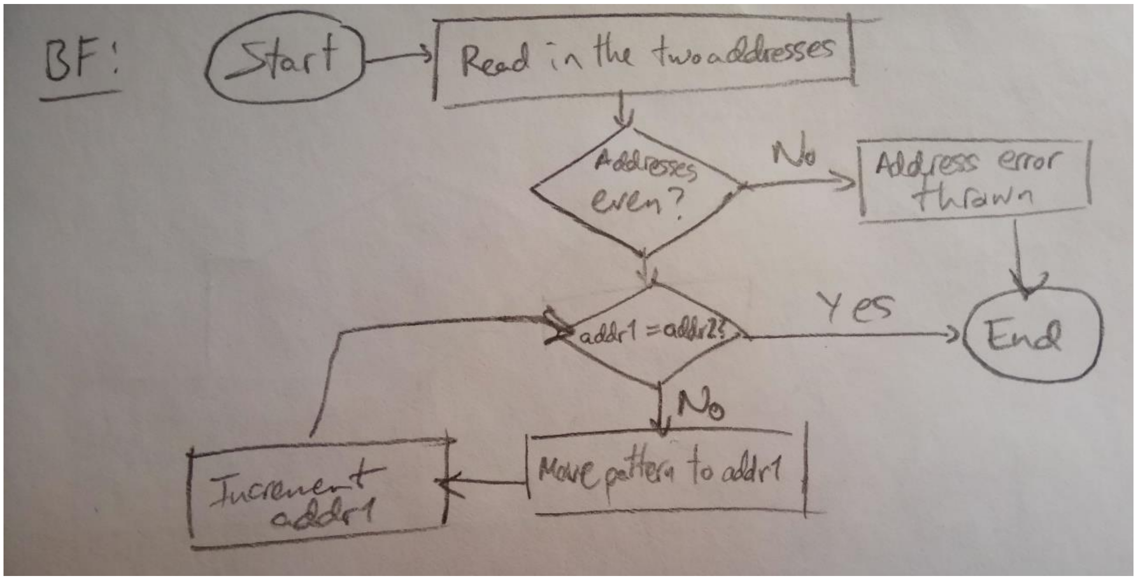

Figure 2.16. Debugger Command #6 Flowchart

2.2.6.2-) Debugger Command #6 Assembly Code

```
* BF -- fills block of memory with word pattern
BF       MOVEM.L D0-D3/D7/A1-A3,-(A7)
         MOVE.B  (A6)+,D1    ; first '$'
         CMPI.B  #$24,D1 ; is it '$'?
         BNE     BFINV ; wrong command usage
         BSR     MEM2HEX ; D1 has 1st address in hex
         MOVEA.L D1,A2   ;store in A2
         MOVE.B  (A6)+,D1    ; space in between addresses
         CMPI.B  #$20,D1 ; is it ' '?
         BNE     BFINV
         MOVE.B  (A6)+,D1    ; second '$'
         CMPI.B  #$24,D1
         BNE     BFINV
         BSR     MEM2HEX ; D1 has 2nd address in hex
         MOVEA.L D1,A3   ; both addresses have been read now
         CLR.L   D2      ; pattern will go in here
         MOVE.B  (A6)+,D1    ; space before the pattern
         CMPI.B  #$00,D1 ; no pattern given, use default
         BEQ     BFSTART
         CMPI.B  #$20,D1 ; is it ' '?
         BNE     BFINV
         MOVE.L  #3,D3   ; counter for remaining 3 digits (if there)
BFPATT   MOVE.B  (A6)+,D7    ; first byte of pattern
         TST.B   D7
         BEQ     BFSTART ; only one digit was given, use first one padded with a zero
         ASL.L   #4,D2   ; place first digit on the left part of the byte
         BSR     ASCII2NUM
         ADD.B   D7,D2   ; goes into the right part of the byte
         DBF     D3,BFPATT   ; debrease D3 and keep looping until all digits read
BFSTART MOVE.W  (A3),D3 ; TEST: if address2 not even, address error is raised
BFLOOP   CMPA.L  A2,A3
         BLE     BFDONE  ; done when A2 reaches A3
         MOVE.W  D2,(A2)+    ; write the pattern in memory. Address error raised if address1 not even
         BRA     BFLOOP
BFINV    BSR     INVALID
BFDONE   MOVEM.L (A7)+,D0-D3/D7/A1-A3
         RTS
```
Figure 2.17. Debugger Command #6 Assembly Code

2.2.7-) Debugger Command #7: BMOV

Moves a block of memory from address1.1 (inclusive) to address1.2 (exclusive) to another place in memory starting at address2.

Note that if address1.1 ≤ address2 < address1.2, all data between address2 and address1.2 will be lost because the data between address1.1 and address2 will be repeatedly copied over at that other memory space.

2.2.7.1-) Debugger Command #7 Algorithm and Flowchart

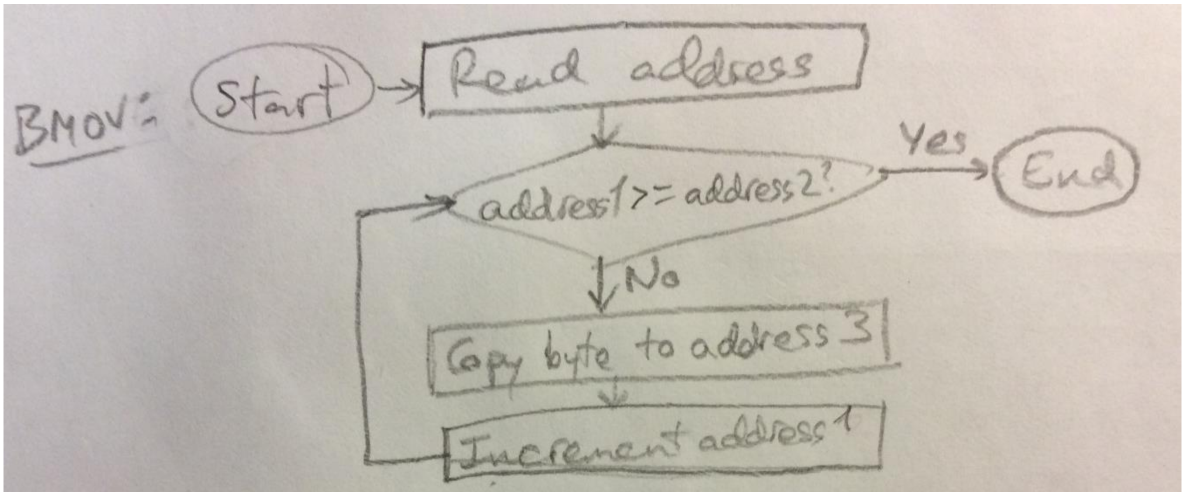

Figure 2.18. Debugger Command #7 Assembly Code

2.2.7.2-) Debugger Command #7 Assembly Code

```
* BMOV -- copies block of memory somewhere else
BMOV     MOVEM.L D1/A2-A4,-(A7)
         MOVE.B  (A6)+,D1     ; first '$'
         CMPI.B  #$24,D1 ; is it '$'?
         BNE     BMINV ; wrong command usage
         BSR     MEM2HEX ; D1 has 1st address in hex
         MOVEA.L D1,A2    ;store in A2
         MOVE.B  (A6)+,D1     ; space in between addresses
         CMPI.B  #$20,D1 ; is it ' '?
         BNE     BMINV
         MOVE.B  (A6)+,D1     ; second '$'
         CMPI.B  #$24,D1
         BNE     BMINV
         BSR     MEM2HEX ; D1 has 2nd address in hex
         MOVE.L  D1,A3    ; store in A3
         MOVE.B  (A6)+,D1     ; space in between addresses
         CMPI.B  #$20,D1 ; is it ' '?
         BNE     BMINV
         MOVE.B  (A6)+,D1     ; third '$'
         CMPI.B  #$24,D1
         BNE     BMINV
         BSR     MEM2HEX ; D1 has 3rd address in hex
         MOVE.L  D1,A4        ; store in A4
BMLOOP   CMPA.L  A2,A3
         BLE     BMDONE  ; done when A2 reaches A3
         MOVE.B  (A2)+,(A4)+ ; copy
         BRA     BMLOOP
BMINV    BSR     INVALID
BMDONE   MOVEM.L (A7)+,D1/A2-A4
         RTS
```

Figure 2.19. Debugger Command #7 Assembly Code

2.2.8-) Debugger Command #8: BTST

Tests all bits between address1 (inclusive) and address2 (exclusive). This is done by writing and reading the patterns $AA and $55 byte by byte, thus changing each bit. An error is raised and displayed if something else is read after writing.

2.2.8.1-) Debugger Command #8 Algorithm and Flowchart

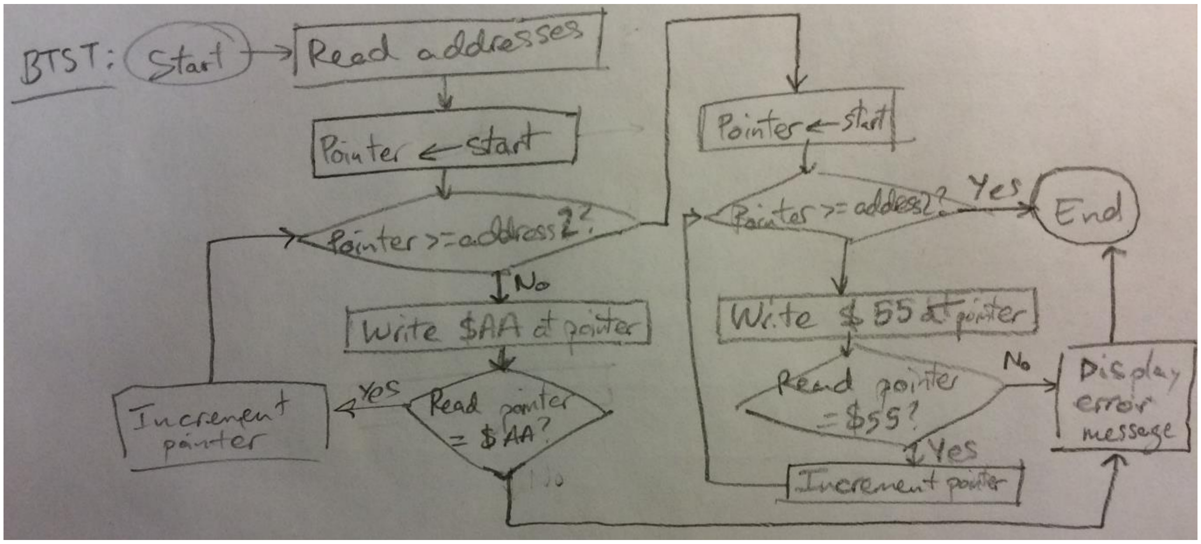

Figure 2.20. Debugger Command #8 Flowchart

2.2.8.2-) Debugger Command #8 Assembly Code

```
* BTST -- tests each bit (by setting and unsetting all) in a block of memory
BTERROR DC.B    'MEMORY ERROR FOUND AT LOCATION $00000000'
BTLOC   DC.B    $A,$D ; this and BTREAD point after for HEX2MEM to work
        DC.B    'Value expected: '
BTEXP   DC.B    '00',$A,$D
        DC.B    'Value read: 00'
BTREAD  DC.B    0
BTST    MOVEM.L D0-D1/A1-A3,-(A7)
        MOVE.B  (A6)+,D1    ; first '$'
        CMPI.B  #$24,D1 ; is it '$'?
        BNE     BTINV ; wrong command usage
        BSR     MEM2HEX ; D1 has 1st address in hex
        MOVEA.L D1,A2   ; store in A2
        MOVEA.L A2,A1   ; store copy for BTLOOP2
        MOVE.B  (A6)+,D1    ; space in between addresses
        CMPI.B  #$20,D1 ; is it ' '?
        BNE     BTINV
        MOVE.B  (A6)+,D1    ; second '$'
        CMPI.B  #$24,D1
        BNE     BTINV
        BSR     MEM2HEX ; D1 has 2nd address in hex
        MOVE.L  D1,A3   ; store in A3
```

```
        CLR.L   D1  ; needed to only look at bytes
BTLOOP1 CMPA.L  A2,A3   ; this loop tries bit pattern 1010
        BLE     BTPRELOOP2
        MOVE.B  #$AA,(A2)   ; write
        MOVE.B  (A2)+,D1    ; read
        CMPI.B  #$AA,D1     ; check correct
        BEQ     BTLOOP1     ; move to next byte
        LEA     BTREAD,A1   ; if here, there is a problem in memory!
        BSR     HEX2MEM_NOZ ; load everything to memory, to be able to print error
        LEA     BTEXP,A1
        MOVE.B  #'A',(A1)+
        MOVE.B  #'A',(A1)
        LEA     BTLOC,A1
        SUBA.L  #1,A2
        MOVE.L  A2,D1
        BSR     HEX2MEM
        LEA     BTERROR,A1
        MOVE.B  #13,D0
        TRAP    #15     ; print the error message
        BRA     BTDONE  ; stop execution
BTPRELOOP2  MOVEA.L A1,A2   ; copy was stored a while back to be able to start over
BTLOOP2 CMPA.L  A2,A3   ; this loop tries bit pattern 0101. Works the same as BTLOOP1
        BLE     BTDONE
        MOVE.B  #$55,(A2)   ; write
        MOVE.B  (A2)+,D1    ; read
        CMPI.B  #$55,D1     ; check correct
        BEQ     BTLOOP2     ; move to next byte
        LEA     BTREAD,A1   ; error in memory, act like before
        BSR     HEX2MEM_NOZ
        LEA     BTEXP,A1
        MOVE.B  #'5',(A1)+
        MOVE.B  #'5',(A1)
        LEA     BTLOC,A1
        SUBA.L  #1,A2
        MOVE.L  A2,D1
        BSR     HEX2MEM
        LEA     BTERROR,A1
        MOVE.B  #13,D0
        TRAP    #15
        BRA     BTDONE
BTINV   BSR     INVALID
BTDONE  MOVEM.L (A7)+,D0-D1/A1-A3
        RTS
```

Figure 2.21. Debugger Command #8 Assembly Code

2.2.9-) Debugger Command #9: BSCH

Searches for an ASII string in a block of memory between address1 (inclusive) and address2 (exclusive).

2.2.9.1-) Debugger Command #9 Algorithm and Flowchart

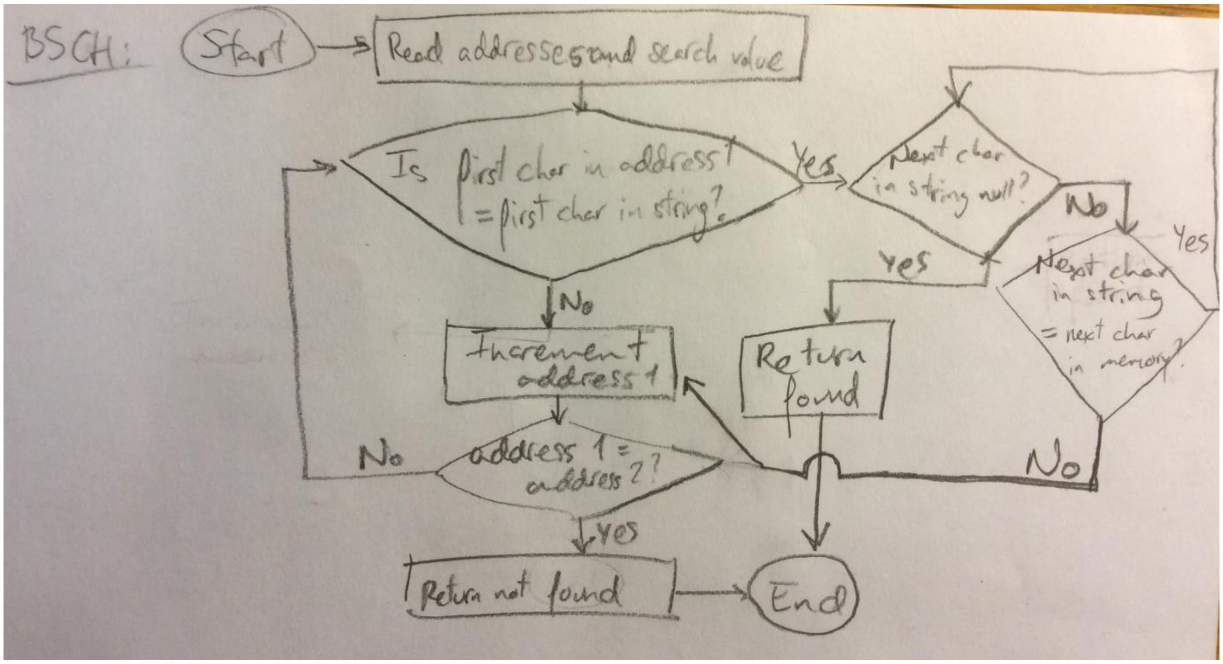

Figure 2.22. Debugger Command #9 Flowchart

2.2.9.2-) Debugger Command #9 Assembly Code

```
* BSCH -- search for string literal in memory block
BSNO        DC.B    'Not found',0
BSYES       DC.B    'Found at location: $00000000'
BSYESADDR   DC.B 0
BSCH    MOVEM.L D1/A1,-(A7)
        LEA     BSNO,A1 ; will change if found
        MOVE.B  (A6)+,D1    ; first '$'
        CMPI.B  #'$',D1 ; is it '$'?
        BNE     BSINV   ; wrong command usage
        BSR     MEM2HEX ; D1 has 1st address in hex
        MOVEA.L D1,A2   ; store in A2
        MOVE.B  (A6)+,D1    ; space in between addresses
        CMPI.B  #' ',D1 ; is it ' '?
        BNE     BSINV
        MOVE.B  (A6)+,D1    ; second '$'
        CMPI.B  #'$',D1
        BNE     BSINV
        BSR     MEM2HEX ; D1 has 2nd address in hex
        MOVE.L  D1,A3   ; store in A3
        MOVE.B  (A6)+,D1    ; a space
        CMPI.B  #' ',D1
        BNE     BSINV
BSLOOP  CMPA.L A2,A3
        BEQ     BSDONE  ; stop if A2 reaches A3 (not found)
        MOVEA.L A6,A4   ; keep A6 for reference
```

```
        CMP.B   (A2)+,(A4)+ ; compare first char
        BNE     BSLOOP  ; look at next if different
        MOVE.L  A2,A5   ; keep A2 for reference
BSMAYB  CMPI.B  #0,(A4) ; see if we reached end of string
        BEQ     BSFOUND ; if we did, the whole string matched!
        CMP.B   (A5)+,(A4)+ ; else, compare next char
        BNE     BSLOOP  ; if not equal, have to check next possible word start
        BRA     BSMAYB  ; if equal, keep on looking in this word
BSINV   BSR     INVALID
        BRA     BSEND
BSFOUND MOVE.L  A2,D1   ; to tell where it was found
        SUBQ.L  #1,D1   ; was off by one
        LEA     BSYESADDR,A1
        BSR     HEX2MEM ; write address in the message
        LEA     BSYES,A1
BSDONE  MOVE.B  #13,D0
        TRAP    #15     ; print message: found or not found
BSEND   MOVEM.L (A7)+,D1/A1
        RTS
```

Figure 2.23. Debugger Command #9 Assembly Code

2.2.10-) Debugger Command #10: GO

Executes a program stored in some location in memory.

2.2.10.1-) Debugger Command #10 Algorithm and Flowchart

GO
Read starting address from input
Jump to that subroutine // execute user's program
Finish

Figure 2.13. Debugger Command #10 Flowchart

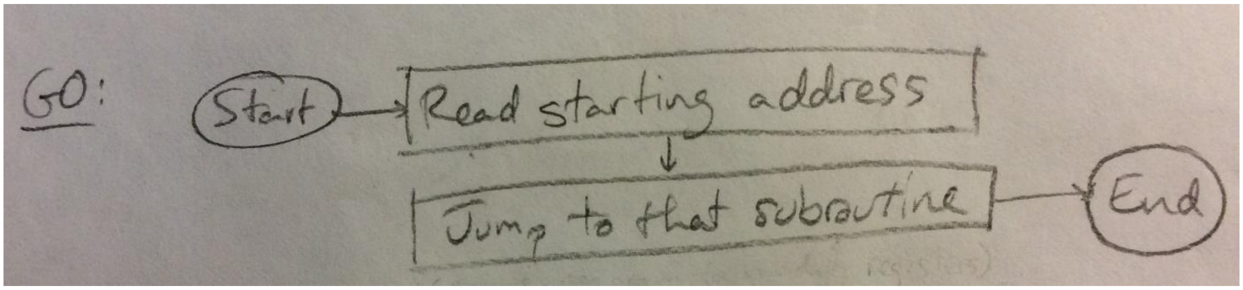

Figure 2.24. Debugger Command #10 Flowchart

2.2.10.2-) Debugger Command #10 Assembly Code

```
* GO -- executes another program
GO      MOVEM.L D0-D7/A0-A7,-(A7)    ; don't allow the program to change registers
        MOVE.B  (A6)+,D1     ; '$'
        CMPI.B  #$24,D1 ; is it '$'?
        BNE     GOINV   ; wrong command usage
        BSR     MEM2HEX ; D1 has address in hex
        MOVEA.L D1,A0   ;store in A0
        JSR     (A0)    ; execute the program
        BRA     GODONE
GOINV   BSR     INVALID
GODONE  MOVEM.L (A7)+,D0-D7/A0-A7
        RTS
```

Figure 2.25. Debugger Command #10 Assembly Code

2.2.11-) Debugger Command #11: DF

Displays all registers as they were before running the monitor program.

2.2.11.1-) Debugger Command #11 Algorithm and Flowchart

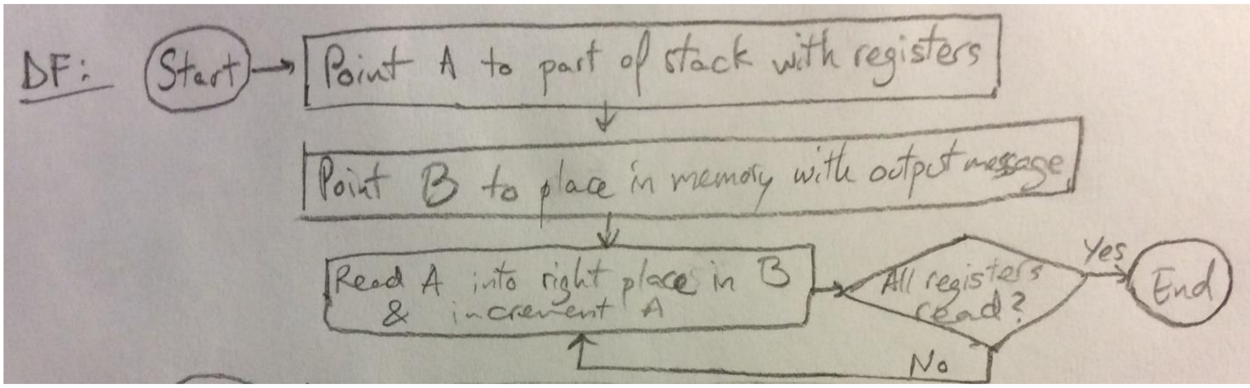

Figure 2.26. Debugger Command #11 Flowchart

2.2.11.2-) Debugger Command #11 Assembly Code

```
* DF -- displays formatted registers
DF      MOVEM.L D0-D2/A0-A1,-(A7)
        LEA     STACK,A0
        ADDA.L  #4,A0   ; placed after A7 in stack
        LEA     DF_MSG_END,A1
DFLOOP  SUBQ.L  #1,A1   ; pass the $A at end of each line
        MOVE.L  #3,D2   ; number of registers per line - 1
DFLINE  MOVE.L  -(A0),D1     ; put register value in D1
        BSR     HEX2MEM      ; will store D1 in -8(A1)
        SUBQ.L  #4,A1   ; skip other characters
        DBF     D2,DFLINE    ; keep looping till line done
        CMP.L   #DF_MSG,A1
        BGT     DFLOOP
```

```
        ADDQ.L   #1,A1    ; put back at the front of the message
        MOVE.B   #13,D0
        TRAP     #15      ; print register value
        MOVEM.L (A7)+,D0-D2/A0-A1
        RTS
```

Figure 2.27. Debugger Command #11 Assembly Code

2.2.12-) Debugger Command #12: EXIT

Terminates the program and restores the registers to the original values.

2.2.12.1-) Debugger Command #12 Algorithm and Flowchart

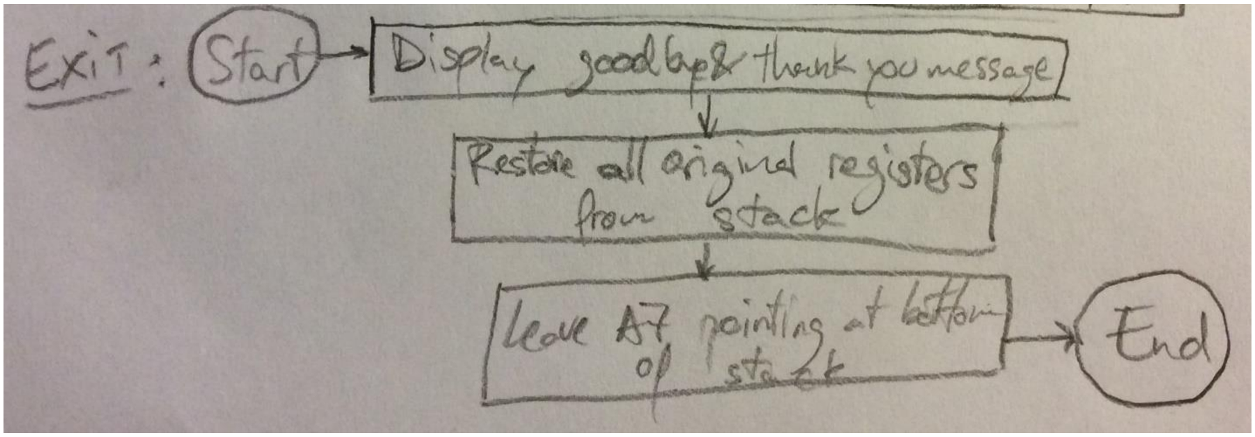

Figure 2.28. Debugger Command #12 Flowchart

2.2.12.2-) Debugger Command #12 Assembly Code

```
* EXIT -- terminates the program
EXIT    LEA      GOODBYE,A1
        MOVE.B   #13,D0
        TRAP     #15      ; print goodbye message
        ADDA.L   #4,A7    ; move past the PC stored in the stack
        ADDA.L   #MAX_IN_LEN,A7  ; move stack back to position prior to reading input
        MOVEM.L (A7)+,D0-D7/A0-A6   ; restore all registers in stack
        MOVEA.L STACK,A7
        BRA      END      ; exit program
```

Figure 2.29. Debugger Command #12 Assembly Code

2.2.13-) Debugger Command #13: BPRINT

Displays an ascii stored in memory to the console. User may provide an ending address (exclusive) or not. If not provided, the string will be terminated when a null character is found in memory. Starting address is inclusive.

2.2.13.1-) Debugger Command #13 Algorithm and Flowchart

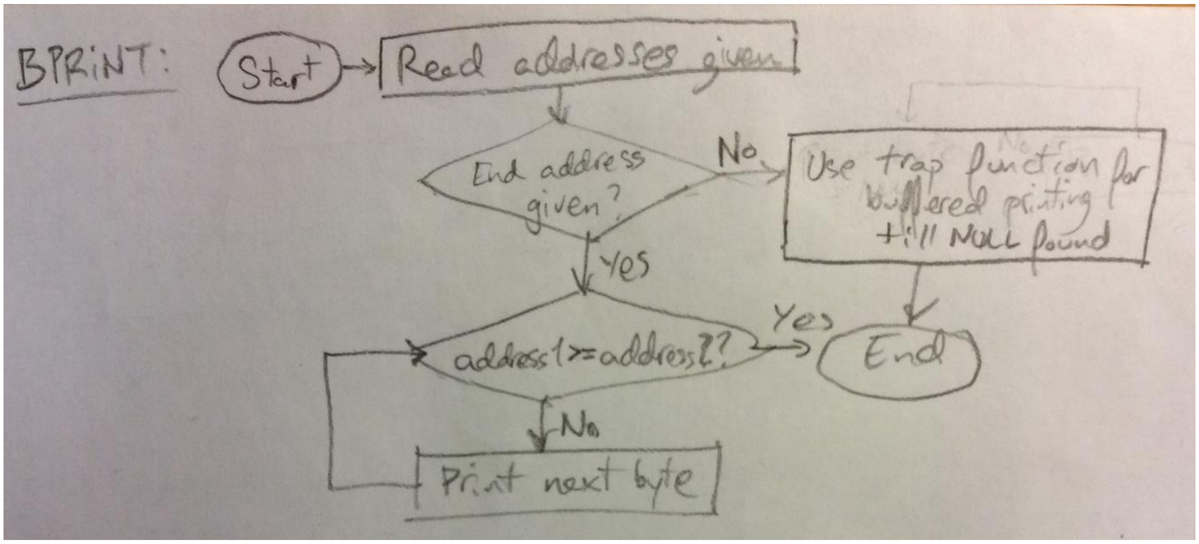

Figure 2.30. Debugger Command #13 Flowchart

2.2.13.2-) Debugger Command #13 Assembly Code

```
* BPRINT -- print as ascii a memory block
BPRINT  MOVEM.L D0-D1/A1-A3,-(A7)
        MOVE.B  (A6)+,D1     ; first '$'
        CMPI.B  #'$',D1 ; is it '$'?
        BNE     BPINV ; wrong command usage
        BSR     MEM2HEX ; D1 has 1st address in hex
        MOVEA.L D1,A2    ; store in A2
        MOVE.B  (A6)+,D1     ; space in between addresses
        CMPI.B  #0,D1    ; is it null?
        BEQ     BPNULL   ; read until null character found
        CMPI.B  #' ',D1 ; is it ' '?
        BNE     BPINV
        MOVE.B  (A6)+,D1     ; second '$'
        CMPI.B  #'$',D1
        BNE     BPINV
        BSR     MEM2HEX ; D1 has 2nd address in hex
        MOVE.L  D1,A3    ; store in A3
        MOVEA.L A6,A1    ; print from here
        MOVE.B  #0,1(A1)     ; make sure
        MOVE.B  #14,D0   ; for printing trap
BPBLOCK CMPA.L  A2,A3
        BLE     BPBDONE ; stop when A2 reaches A3
        MOVE.B  (A2)+,(A1)   ; put byte in (A1)
        TRAP    #15 ; print that byte!
        BRA     BPBLOCK
BPBDONE MOVE.B  #0,(A1)
        MOVE.B  #13,D0
        TRAP    #15      ; print a line feed and carriage return
        BRA     BPDONE
```

```
BPNULL  MOVEA.L A2,A1    ; no limit given, so print till null char found
        MOVE.B  #13,D0
        TRAP    #15      ; print!
        BRA     BPDONE
BPINV   BSR     INVALID
BPDONE  MOVEM.L (A7)+,D0-D1/A1-A3
        RTS
```

Figure 2.31. Debugger Command #13 Assembly Code

2.2.14-) Debugger Command #14: CONV

Converts a hex value to (preceded by a dollar sign '$') to decimal, and vice versa.

2.2.14.1-) Debugger Command #14 Algorithm and Flowchart

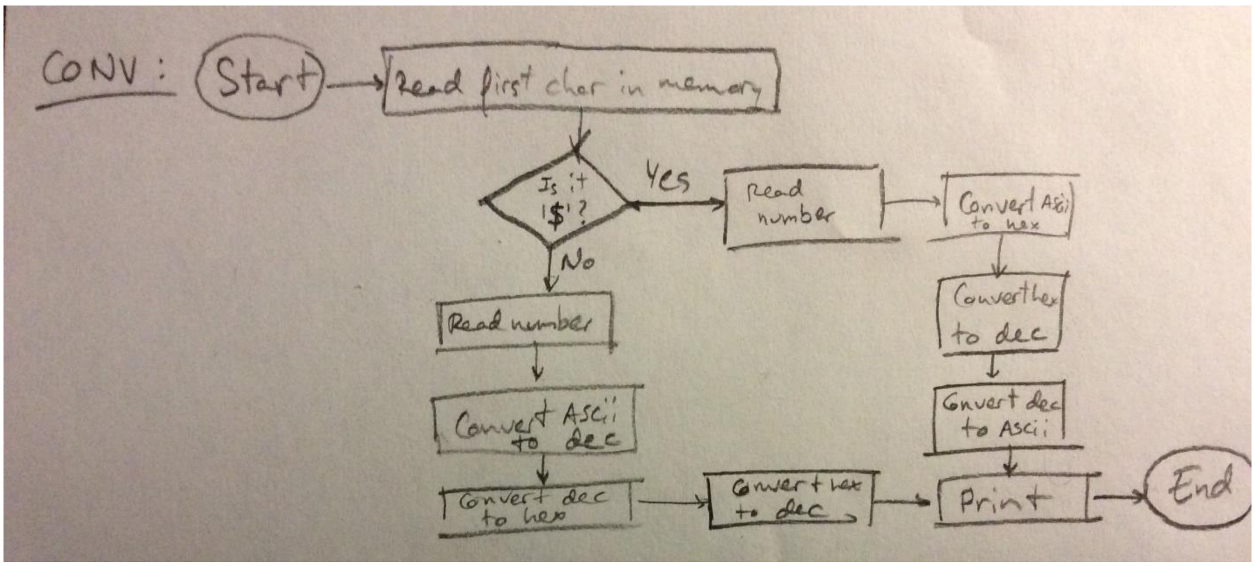

Figure 2.32. Debugger Command #14 Flowchart

2.2.14.2-) Debugger Command #14 Assembly Code

```
* CONV -- takes in hex and returns decimal, or viceversa
CONV     MOVEM.L D0-D1/A1,-(A7)
         MOVE.B  (A6)+,D1
         CMPI.B  #$24,D1 ; is it '$'?
         BEQ     CONVH2D ; if so, hex to dec
CONVD2H  SUBQ.L  #1,A6    ; point back at first number
         BSR     MEM2DEC ; D1 contains the decimal number
         MOVEA.L A6,A1    ; number ready to print
         BSR     HEX2MEM_NOZ ; that number is written as hex in memory
         MOVE.B  #'$',-(A1)
         BRA     CONVDONE
CONVH2D  BSR     MEM2HEX ; convert ascii to hex
         MOVEA.L A6,A1    ;number ready to print
         BSR     DEC2MEM ; convert it back to ascii but as decimal
CONVDONE MOVE.B #13,D0
         TRAP    #15 ; print result
         MOVEM.L (A7)+,D0-D1/A1
         RTS
```

Figure 2.33. Debugger Command #14 Assembly Code

2.3-) Exception Handlers

All the exceptions accounted for in this program are:

- Address Error
- Bus Error
- Illegal Instruction
- Privilege Error
- Division By Zero
- Check Error
- Line A Emulator
- Line F Emulator

Since the only difference amongst all exception handlers is the display message identifying the error, they were implemented with common code, except for the message itself. As it is specified in the next sub-sections, all handlers load the appropriate message, and then allow for the common code to also call DF for printing the registers and return to the main routine appropriately.

Nevertheless, the Bus Error and Address Error handlers do differ from the rest in that they also display the SSW, BA and IR. This is added as extra code for these two handlers only.

2.3.1-) Exception Handler Algorithm and Flowchart

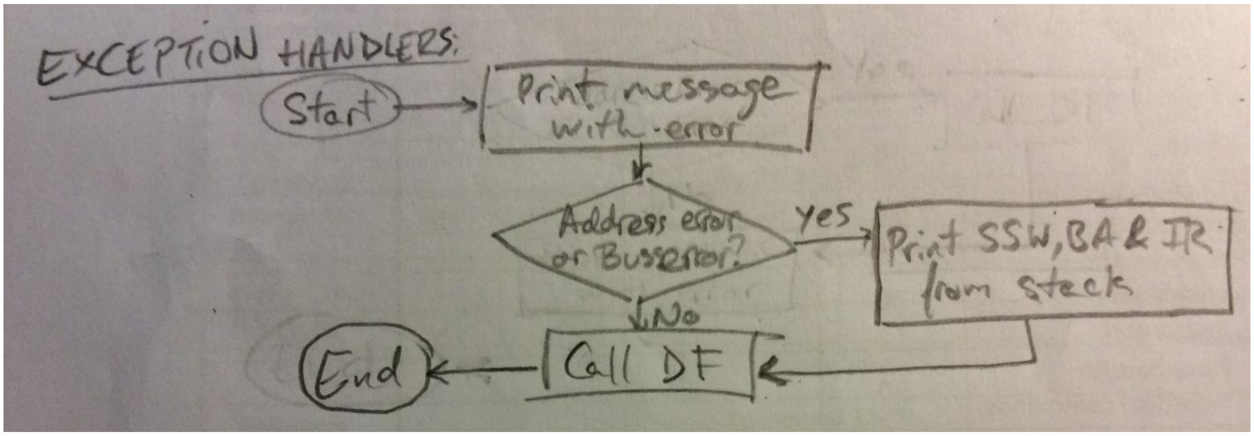

Figure 2.34. Debugger Command # 1 Flowchart

2.3.2-) Exception Handler Assembly Code

The assembly code for all handlers:

```
*** EXCEPTION HANDLERS ***
ADDRERR MOVEM.L D0/A1,-(A7)
        MOVEM.L D1/A0,-(A7) ; specific for this interrupt
        LEA     ADDRERR_MSG,A1
        MOVE.B  #13,D0
        TRAP    #15
        BRA     INTERR_REG  ; print the special registers
BERR    MOVEM.L D0/A1,-(A7)
        MOVEM.L D1/A0,-(A7) ; specific for this interrupt
        LEA     BERR_MSG,A1
        MOVE.B  #13,D0
        TRAP    #15
        BRA     INTERR_REG  ; print the special registers
ILLINS  MOVEM.L D0/A1,-(A7)
        LEA     ILLINS_MSG,A1
        BRA     INTERR
PRIVERR MOVEM.L D0/A1,-(A7)
        LEA     PRIVERR_MSG,A1
        BRA     INTERR
DIV0    MOVEM.L D0/A1,-(A7)
        LEA     DIV0_MSG,A1
        BRA     INTERR
CHKERR  MOVEM.L D0/A1,-(A7)
        LEA     CHKERR_MSG,A1
        BRA     INTERR
LINEA   MOVEM.L D0/A1,-(A7)
        LEA     LINEA_MSG,A1
        BRA     INTERR
LINEF   MOVEM.L D0/A1,-(A7)
        LEA     LINEF_MSG,A1
        BRA     INTERR
```

```
INTERR_REG  ; only BERR and ADDRERR do this
        MOVEA.L A7,A0
        ADDA.L  #24,A0  ; A0 is pointing right below SSW, BA and IR
        MOVEA.L #STACK,A1
        SUBA.L  #60,A1  ; write message in the input space of the stack (currently unused)
        MOVE.B  #0,-(A1)    ; null terminator
        CLR.L   D1
        MOVE.W  -(A0),D1    ; SSW in D1
        BSR     HEX2MEM
        ADDQ.L  #4,A1       ; only want SSW to be a word
        MOVE.B  #' ',-(A1)
        MOVE.L  -(A0),D1    ; BA in D1
        BSR     HEX2MEM
        MOVE.B  #' ',-(A1)
        CLR.L   D1
        MOVE.W  -(A0),D1    ; IR in D1
        BSR     HEX2MEM
        ADDQ.L  #4,A1       ; only want IR to be a word
        MOVEM.L (A7)+,D1/A0 ; restore these specific registers
INTERR  MOVE.B  #13,D0
        TRAP    #15 ; print corresponding message for that interrupt
        BSR     DF  ; print registers
        MOVEM.L (A7)+,D0/A1 ; do here to be able to modify values of A7
        LEA     STACK,A7    ; next 3 instructions put A7 at beginning of input space in stack
        SUBA.L  #60,A7  ; 15 registers that occupy 4 bytes each (2*4 = 8 bits)
        SUBA.L  #MAX_IN_LEN,A7  ; the input space
        BRA     PROMPT
```

Figure 2.35. Exception Handling Routines Assembly Code

2.4-) Quick User Instruction Manual

The following text contains a quick user manual with usage for all commands. All addresses must be given in hex, and all hex values must be given with a preceding dollar sign ‘$’. It can be accessed from the running program by executing the ‘HELP’ command:

HELP: Displays This Message

MDSP: Outputs Address And Memory Contents
Default address2: address1 + 16
MDSP <address1>[<address2>] eg: MDSP $908 $904<CR>

SORTW: Sorts Unsigned Words In A Memory Block
Both address1 and address2 are inclusive
Default order: descending
SORTW <address1> <address2>[A|D] eg: SORTW $2000 $201E A<CR>

MM: Modifies Data In Memory
Default: Displays one byte
W: Displays one word
L: Displays one long word
MM <address>[size]

MS: Set Memory To Given ASCII Or Hex
Default: ASCII. Prepend $ for hex (byte, word or long)
MS <address1> [$]<string|hex> eg: MS $4000 Hello!

BF: Fills Block Of Memory With Word Pattern
Both addresses must be even
Default pattern: 0000
If less than 4 digits given, right justified and zero padded
BF <address1> <address2>[pattern] eg: BF $2000 $2200 4325<CR>
BMOV: Duplicate A Memory Block At Another Address
Must provide two addresses (inclusive, exclusive) for first block
Only one address (inclusive start) for second block
BMOV <address1.1> <address1.2> <address2>

BTST: Test Memory Block
BTST <address1> <address2>

BSCH: Search In Memory Block
BSCH <address1> <address2> <string>

GO: Execute Another Program
GO <address1>

DF: Displays All Formatted Registers eg: DF<CR>

EXIT: Exit The Monitor Program eg: EXIT<CR>

The two extra commands:

BPRINT: Print Block Of Memory
Default end: wherever a null char is found
BPRINT <address1>[<address2>]

CONV: Convert Hex to Decimal, Or Vice Versa
CONV [$]num eg: CONV 16<CR> returns $10

2.4.1-) Assembly Code

The above message is stored in memory between locations $10FE and $16FB (~1.5KB). The assembly code for it is shown on the next page:

```
HELP_MSG    DC.B    'HELP: Displays This Message',$A,$A,$D
            DC.B    'MDSP: Outputs Address And Memory Contents',$A,$D
            DC.B    'Default address2: address1 + 16',$A,$D
            DC.B    'MDSP <address1>[ <address2>] eg: MDSP $908 $904<CR>',$A,$A,$D
            DC.B    'SORTW: Sorts Unsigned Words In A Memory Block',$A,$D
            DC.B    'Both address1 and address2 are inclusive',$A,$D
            DC.B    'Default order: descending',$A,$D
            DC.B    'SORTW <address1> <address2>[ A|D] eg: SORTW $2000 $201E A<CR>',$A,$A,$D
            DC.B    'MM: Modifies Data In Memory',$A,$D
            DC.B    'Default: Displays one byte',$A,$D
            DC.B    'W: Displays one word',$A,$D
            DC.B    'L: Displays one long word',$A,$D
            DC.B    'MM <address>[ size]',$A,$A,$D
            DC.B    'MS: Set Memory To Given ASCII Or Hex',$A,$D
            DC.B    'Default: ASCII. Prepend $ for hex (byte, word or long)',$A,$D
            DC.B    'MS <address1> [$]<string|hex> eg: MS $4000 Hello!',$A,$A,$D
            DC.B    'BF: Fills Block Of Memory With Word Pattern',$A,$D
            DC.B    'Both addresses must be even',$A,$D
            DC.B    'Default pattern: 0000',$A,$D
            DC.B    'If less than 4 digits given, right justified and zero padded',$A,$D
            DC.B    'BF <address1> <address2>[ pattern] eg: BF $2000 $2200 4325<CR>',0
HELP_MSG2   DC.B    'BMOV: Duplicate A Memory Block At Another Address',$A,$D
            DC.B    'Must provide two addresses (inclusive, exclusive) for first block',$A,$D
            DC.B    'Only one address (inclusive start) for second block',$A,$D
            DC.B    'BMOV <address1.1> <address1.2> <address2>',$A,$A,$D
            DC.B    'BTST: Test Memory Block',$A,$D
            DC.B    'BTST <address1> <address2>',$A,$A,$D
            DC.B    'BSCH: Search In Memory Block',$A,$D
            DC.B    'BSCH <address1> <address2> <string>',$A,$A,$D
            DC.B    'GO: Execute Another Program',$A,$D
            DC.B    'GO <address1>',$A,$A,$D
            DC.B    'DF: Displays All Formatted Registers eg: DF<CR>',$A,$A,$D
            DC.B    'EXIT: Exit The Monitor Program eg: EXIT<CR>',$A,$A,$D
            DC.B    'The two extra commands:',$A,$A,$D
            DC.B    'BPRINT: Print Block Of Memory',$A,$D
            DC.B    'Default end: wherever a null char is found',$A,$D
            DC.B    'BPRINT <address1>[ <address2>]',$A,$A,$D
            DC.B    'CONV: Convert Hex to Decimal, Or Vice Versa',$A,$D
            DC.B    'CONV [$]num eg: CONV 16<CR> returns $10',0
```

Figure 2.13. Help Message Assembly Code

3-) Discussion

The design of this monitor program involved a lot of decision taking, much of which can be encountered by a professional engineer almost in a daily basis. Keeping in mind the main goal of producing a fully functional program, many other optimization factors had to be considered throughout both the design and implementation processes. These factors included:

- Speed
- Memory usage
- Simplicity
- Readability of code
- Usability

An example of a decision that took into consideration most of these factors was the implementation of the exception handling routines. In order to keep the code simple and readable and to cut down significantly the use of memory, the common functionality of all these subroutines was identified and implemented as a subroutine on its own. Thereby, each of the handlers could simply perform their specific function, such as loading their particular display message, and then branch onto the common part of the subroutine. This avoided having various similar subroutines for each exception, which would have occupied more memory unnecessarily.

In fact, modularity was a big part to this project. By creating several helper subroutines, such as those for ASCII to hex conversion and vice versa or the one for displaying an invalid command's message, the code was kept clean and efficient.

Another essential part of the project was the design of the algorithms themselves. Writing pseudocode and flowcharts for each command was definitely good practice in coming up with efficient and well-written code.

In addition, it is worth noting the necessity to consider the user end as well. For this product to be actually useful, it must be usable. Therefore, the format in which the input would be taken was considered carefully and mindfully of the common conventions.

Finally, it must be acknowledged that in this project, as in almost any other production level one, some debugging was required. In this case it was mostly software debugging, but still keeping in mind the hardware, computer architecture and other processor concepts.

4-) Feature Suggestions

Firstly, many other commands may be implemented, depending on the users' needs. Some of these may include loading a program from some other executable file into memory, interchanging memory blocks, etc.

Secondly, the exceptions could be advanced to try to fix the error, or give suggestions on how to fix it, instead of simply acknowledging the error.

Thirdly, the help message could be stored in a more specific region of memory or in external memory, since it occupies a large space and could be an obstacle to other functionality.

Lastly, functionality to interact with peripheral devices could be added.

5-) Conclusion

All commands and exception handling routines were implemented successfully. Any user will be capable of executing the previously described functionality. With the help of the Quick User Instruction Manual, it is not hard to get started quickly.

In addition, error handling has been implemented, so that it is difficult and unexpected for the common user to break the code or come across unforeseen hindrances. As long as the commands are used in a logical manner, following the given descriptions, the program will run appropriately.

6-) References

[1] T. Harman and D. Hein, "The Motorola MC 68000 Microprocessor Family", Prentice-Hall Inc., Englewood Cliffs, NJ, 1996.

[2] A. Clements, "Microprocessor Systems Design", PWS Publishing Company, Boston, MA, 1997.

[3] Educational Computer Board Manual

[4] Experiment 2 Lab Manual

[5] Experiment 3 Lab Manual

```
*-----------------------------------------------------------
* Title      : Monitor Design Project
* Written by : Javier Sorribes
* Date       : 4/2/17
* Description: Allows user to repeteadly run debugging commands.
*              Provides asynchronous exception handling routines.
*-----------------------------------------------------------
*** MEMORY INITIALIZATION ***
    ORG     $1000       ; stack and other memory
; $A is newline, $D carriage return, $20 whitespace
WELCOME     DC.B    'WELCOME TO MONITOR441! - BY JAVIER SORRIBES',0
GOODBYE     DC.B    'THANK YOU FOR USING MONITOR441, SEE YOU SOON!',0
PROMPT_STR  DC.B    $A,$D,'MONITOR441>',0 ; might want to add one space
INVALID_MSG DC.B    'INVALID COMMAND',$A,$D
            DC.B    'Type HELP for command usage',0

COM_TABL    DC.B    '4HELP',0   ; Command names table
            DC.B    '4MDSP',$20  ; number specifies length of word
            DC.B    '5SORTW',$20    ; used for SEARCH (not input)
            DC.B    '2MM',$20
            DC.B    '2MS',$20
            DC.B    '2BF',$20
            DC.B    '4BMOV',$20
            DC.B    '4BTST',$20
            DC.B    '4BSCH',$20
            DC.B    '2GO',$20
            DC.B    '2DF',0
            DC.B    '4EXIT',0
            DC.B    '6BPRINT',$20
            DC.B    '4CONV',$20

COM_ADDR    DC.W    HELP        ; Command addresses table
            DC.W    MDSP
            DC.W    SORTW
            DC.W    MM
            DC.W    MS
            DC.W    BF
            DC.W    BMOV
            DC.W    BTST
            DC.W    BSCH
            DC.W    GO
            DC.W    DF
            DC.W    EXIT
            DC.W    BPRINT
            DC.W    CONV

HELP_MSG    DC.B    'HELP: Displays This Message',$A,$A,$D
            DC.B    'MDSP: Outputs Address And Memory Contents',$A,$D
            DC.B    'Default address2: address1 + 16',$A,$D
            DC.B    'MDSP <address1>[ <address2>] eg: MDSP $908 $904<CR>',$A,$A,$D
            DC.B    'SORTW: Sorts Unsigned Words In A Memory Block',$A,$D
            DC.B    'Both address1 and address2 are inclusive',$A,$D
            DC.B    'Default order: descending',$A,$D
            DC.B    'SORTW <address1> <address2>[ A|D] eg: SORTW $2000 $201E A<CR>',$A,$A,$D
            DC.B    'MM: Modifies Data In Memory',$A,$D
            DC.B    'Default: Displays one byte',$A,$D
            DC.B    'W: Displays one word',$A,$D
            DC.B    'L: Displays one long word',$A,$D
            DC.B    'MM <address>[ size]',$A,$A,$D
            DC.B    'MS: Set Memory To Given ASCII Or Hex',$A,$D
            DC.B    'Default: ASCII. Prepend $ for hex (byte, word or long)',$A,$D
            DC.B    'MS <address1> [$]<string|hex> eg: MS $4000 Hello!',$A,$A,$D
            DC.B    'BF: Fills Block Of Memory With Word Pattern',$A,$D
            DC.B    'Both addresses must be even',$A,$D
            DC.B    'Default pattern: 0000',$A,$D
            DC.B    'If less than 4 digits given, right justified and zero padded',$A,$D
            DC.B    'BF <address1> <address2>[ pattern] eg: BF $2000 $2200 4325<CR>',0
HELP_MSG2   DC.B    'BMOV: Duplicate A Memory Block At Another Address',$A,$D
            DC.B    'Must provide two addresses (inclusive, exclusive) for first block',$A,$D
            DC.B    'Only one address (inclusive start) for second block',$A,$D
            DC.B    'BMOV <address1.1> <address1.2> <address2>',$A,$A,$D
            DC.B    'BTST: Test Memory Block',$A,$D
            DC.B    'BTST <address1> <address2>',$A,$A,$D
            DC.B    'BSCH: Search In Memory Block',$A,$D
            DC.B    'BSCH <address1> <address2> <string>',$A,$A,$D
```

```
            DC.B    'GO: Execute Another Program',$A,$D
            DC.B    'GO <address1>',$A,$A,$D
            DC.B    'DF: Displays All Formatted Registers eg: DF<CR>',$A,$A,$D
            DC.B    'EXIT: Exit The Monitor Program eg: EXIT<CR>',$A,$A,$D
            DC.B    'The two extra commands:',$A,$A,$D
            DC.B    'BPRINT: Print Block Of Memory',$A,$D
            DC.B    'Default end: wherever a null char is found',$A,$D
            DC.B    'BPRINT <address1>[ <address2>]',$A,$A,$D
            DC.B    'CONV: Convert Hex to Decimal, Or Vice Versa',$A,$D
            DC.B    'CONV [$]num eg: CONV 16<CR> returns $10',0

DF_MSG      DC.B    'D0=XXXXXXXX D1=XXXXXXXX D2=XXXXXXXX D3=XXXXXXXX',$A,$D
            DC.B    'D4=XXXXXXXX D5=XXXXXXXX D6=XXXXXXXX D7=XXXXXXXX',$A,$D
            DC.B    'A0=XXXXXXXX A1=XXXXXXXX A2=XXXXXXXX A3=XXXXXXXX',$A,$D
            DC.B    'A4=XXXXXXXX A5=XXXXXXXX A6=XXXXXXXX A7=XXXXXXXX',0
DF_MSG_END

ADDRERR_MSG DC.B    $D,'Address Error Exception',0
BERR_MSG    DC.B    $D,'Bus Error Exception',0
ILLINS_MSG  DC.B    $D,'Illegal Instructor Exception',0
PRIVERR_MSG DC.B    $D,'Privilege Error Exception',0
DIV0_MSG    DC.B    $D,'Division By Zero Exception',0
CHKERR_MSG  DC.B    $D,'Check Exception',0
LINEA_MSG   DC.B    $D,'Line A Exception',0
LINEF_MSG   DC.B    $D,'Line F Exception',0

*** RUNNING PROGRAM ***
    ;ORG     $1200 --> allow for as much as necessary before this, and add program right after it
START:                   ; first instruction of program

MAX_IN_LEN  EQU 80  ; to ensure input won't overflow stack
STACK       EQU $2FFC   ; $3000 minus a long word because A7 will be stored first
        MOVE.L  A7,STACK    ; store original location of stack beforehand
        LEA     STACK,A7
        MOVEM.L D0-D7/A0-A6,-(A7)   ; store all registers in stack. Want to be able to restore them

** Populate exception vector table ***
        MOVE.L  #BERR,$8
        MOVE.L  #ADDRERR,$C
        MOVE.L  #ILLINS,$10
        MOVE.L  #DIV0,$14
        MOVE.L  #CHKERR,$18
        MOVE.L  #PRIVERR,$20
        MOVE.L  #LINEA,$28
        MOVE.L  #LINEF,$2C

*** MAIN: Prompt, execute and repeat ***
        LEA     WELCOME,A1
        MOVE.B  #13,D0
        TRAP    #15     ; display welcome message
        SUBA.L  #MAX_IN_LEN,A7  ; open space in stack for input (do only once)
*** COMMAND INTERPRETER ***
PROMPT  LEA     PROMPT_STR,A1
        MOVE.B  #14,D0
        TRAP    #15     ; print out prompt
        MOVEA.L A7,A1   ; input will go in stack
        MOVE.B  #2,D0
        TRAP    #15     ; read user input, length stored in D1

        LEA     COM_TABL,A4 ; beginning of command table
        LEA     COM_ADDR,A5 ; end of command table
        CLR.L   D3      ; will be the count of where the command is
SEARCH  CLR.L   D2
        MOVE.B  (A4)+,D2   ; length of next command string
        SUBI.B  #$30,D2 ; convert ascii num to hex
        MOVEA.L A1,A6   ; pointer to input string
CMP_B   CMPM.B  (A4)+,(A6)+ ; compare byte to byte with command names
        DBNE    D2,CMP_B    ; keep comparing characters until length is over
        TST.W   D2
        BLT     EXEC    ; loop was exhausted and all chars were equal
        ADDA.L  D2,A4   ; go to end of command
        ADDQ.L  #2,D3   ; else, increment offset by word size
        CMPA.L  A4,A5 ; end of COM_TABL
        BGE     SEARCH  ; keep on searching
```

```
        BSR     INVALID ; print invalid command message
        BRA     PROMPT ; prompt again

EXEC    ADDA.L  D3,A5   ; add offset to COM_ADDR start
        MOVEA.L #0,A3   ; clear A3, used for subroutine call
        MOVEA.W (A5),A3 ; move that command's address to register
        JSR     (A3)    ; jump to that command's subroutine (below)

        BRA     PROMPT  ; prompt again

*** DEBUGGING COMMANDS ***
* HELP -- displays help message
HELP    MOVEM.L D0-D1/A1,-(A7) ; store used registers in stack
        LEA     HELP_MSG,A1
        MOVE.B  #13,D0
        TRAP    #15     ; print first part of the help message
        MOVE.B  #5,D0
        TRAP    #15     ; wait for the user to enter a character
        LEA     HELP_MSG2,A1
        MOVE.B  #13,D0
        TRAP    #15     ; print second half of the message
        MOVEM.L (A7)+,D0-D1/A1 ; restore registers from stack
        RTS

* For this subroutine and others, A6 contains the start of the command's parameters
* eg: MDSP $1230 $1890 <- A6 points to the first '$'

* MDSP -- displays memory block
MDSP    MOVEM.L D0-D4/A1-A4,-(A7)
        MOVE.B  (A6)+,D1    ; first '$'
        CMPI.B  #$24,D1 ; is it '$'?
        BNE     MDSPINV ; wrong command usage
        BSR     MEM2HEX ; D1 has 1st address in hex
        MOVEA.L D1,A2   ;store in A2
        MOVE.B  (A6)+,D1    ; space in between addresses
        TST.B   D1  ; if null, no 2nd address, so address2 = address1 + 16
        BNE     MDSPADDR2
        MOVEA.L A2,A3
        ADDA.L  #16,A3  ; A3 = A2 +16
        BRA     MDSPLOOP
MDSPADDR2   MOVE.B  (A6)+,D1    ; second '$'
        CMPI.B  #$24,D1
        BNE     MDSPINV
        BSR     MEM2HEX ; D1 has 2nd address in hex
        MOVEA.L D1,A3
MDSPLOOP    MOVEA.L A7,A1
        SUBA.L  #$40,A1 ; move A1 far from A7 to avoid collision in subroutines
        MOVE.B  #$00,-(A1) ; null terminator
        MOVE.B  #$20,-(A1)  ; space
        MOVE.B  #$3E,-(A1)  ; '>' for nicer output
        MOVE.L  A2,D1   ; memory address into D1
        BSR     HEX2MEM ; puts digits of D1 into -X(A1) in ascii (no trailing zeros)
        MOVE.B  #$24,-(A1)  ; '$' for nicer output
        MOVE.B  #14,D0
        TRAP    #15     ; print current memory address
        MOVE.B  #$00,-(A1)  ; null terminator
        MOVE.L  (A2)+,D1    ; memory value into D1
        BSR     HEX2MEM ; puts digits of D1 into -X(A1) in ascii (no trailing zeros)
        MOVE.B  #13,D0
        TRAP    #15     ; print
        CMPA.L  A2,A3
        BGT     MDSPLOOP
        BRA     MDSPDONE
MDSPINV BSR     INVALID ; print invalid command message
MDSPDONE    MOVEM.L (A7)+,D0-D4/A1-A4
        RTS

* SORTW -- implements bubble sort (unsigned numbers)
SORTW   MOVEM.L D0-D4/A1-A4,-(A7)
        MOVE.B  (A6)+,D1    ; first '$'
        CMPI.B  #$24,D1     ; is it '$'?
        BNE     SORTWINV    ; wrong command usage
        BSR     MEM2HEX     ; D1 has 1st address in hex
        MOVEA.L D1,A2       ; store in A2
        MOVE.B  (A6)+,D1    ; space in between addresses
```

```
        CMPI.B  #$20,D1     ; is it ' '?
        BNE     SORTWINV    ; wrong command usage
        MOVE.B  (A6)+,D1    ; second '$'
        CMPI.B  #$24,D1     ; is it '$'?
        BNE     SORTWINV    ; wrong command usage
        BSR     MEM2HEX     ; D1 has now the 2nd address
        MOVEA.L D1,A3       ; store in A3
        MOVE.B  (A6)+,D1    ; space
        CMPI.B  #$00,D1     ; is it NULL?
        BEQ     SORTWDEF    ; use default: descending (D1=0)
        CMPI.B  #$20,D1     ; or is it ' '?
        BNE     SORTWINV    ; wrong command usage
        MOVE.B  (A6)+,D1    ; char either 'A' or 'D'
        CMPI.B  #$41,D1     ; is it 'A'?
        BEQ     SORTWLOOP   ; if so, D1 marks ascending
        CMPI.B  #$44,D1     ; else, is it 'D'?
        BNE     SORTWINV    ; if it isn't, input was invalid
SORTWDEF    CLR.L   D1          ; if it is, D1=0 marks descending
SORTWLOOP   MOVEA.L A2,A4   ; first address copied into A4
SORTWCMP    TST.B   D1      ; tells us whether ascending or descending
        BEQ     SORTWD  ; do descending
SORTWA  CMP.W   (A4)+,(A4)+ ; compare next two numbers
        BCS     SORTWSWAP   ; swap if not in ascending order (if 1st>2nd)
        BRA     SORTWNEXT   ; otherwise, move on
SORTWD  CMP.W   (A4)+,(A4)+ ; compare next two numbers
        BHI     SORTWSWAP   ; swap if not in descending order (if 2nd>1st)
SORTWNEXT   SUBQ.L  #2,A4   ; look back at previous number
        CMP.L   A4,A3
        BNE     SORTWCMP    ; keep comparing if not at end yet (A3 inclusive)
        BRA     SORTWDONE   ; else, done
SORTWSWAP MOVE.L  -(A4),D4   ; move both words to register
        SWAP.W  D4  ; swap the two words
        MOVE.L  D4,(A4) ; write them back
        BRA     SORTWLOOP   ; loop again from start
SORTWINV    BSR INVALID
SORTWDONE   MOVEM.L (A7)+,D0-D4/A1-A4
        RTS

* MM -- modifies data in memory. Size can be B, W or L
MM      MOVEM.L D0-D1/A0-A1,-(A7)
        MOVEA.L A6,A1   ; A1 used for I/O later
        MOVE.B  (A6)+,D1    ; '$'
        CMPI.B  #$24,D1 ; is it '$'?
        BNE     INVALID ; wrong command usage
        BSR     MEM2HEX ; D1 has address in hex
        MOVEA.L D1,A0   ;store in A0
        MOVE.B  (A6)+,D1    ; ' ' before option
        CMPI.B  #0,D1       ; is it null?
        BEQ     MMBYTE  ; use default: byte
        CMPI.B  #$20,D1 ; is it ' '?
        BNE     INVALID ; wrong command usage
        MOVE.B  (A6)+,D1    ; the option
        CMPI.B  #'B',D1
        BEQ     MMBYTE
        CMPI.B  #'W',D1
        BEQ     MMWORD
        CMPI.B  #'L',D1
        BEQ     MMLONG
        BRA     MMINV   ; wrong option
MMBYTE  ADDA.L  #14,A1  ; output will be 13 chars long + null
        MOVE.B  #0,-(A1)    ; null terminator
        MOVE.B  #'?',-(A1)  ; nicer output
        CLR.L   D1
        MOVE.B  (A0),D1     ; content of memory to D1
        BSR     HEX2MEM     ; writes memory content to -8(A1)
        ADDA.L  #6,A1       ; we only want 2 chars, not 8
        MOVE.B  #$9,-(A1)   ; a tabspace
        MOVE.L  A0,D1       ; memory address
        BSR     HEX2MEM     ; memory address to -8(A1)
        MOVE.B  #'$',-(A1)  ; nicer output
        MOVE.B  #14,D0
        TRAP    #15         ; print
        MOVE.B  #2,D0
        TRAP    #15         ; read new value, if any
        CMPI.B  #0,(A1)
```

```
        BNE     MMBNEXT     ; skip memory address?
        ADDA.L  #1,A0       ; if yes, increment A0
        BRA     MMBYTE      ; ...and loop
MMBNEXT CMPI.B  #'.',(A1)   ; else, check if done (entered '.')
        BEQ     MMDONE
        MOVEA.L A1,A6       ; new value to write in!
        BSR     MEM2HEX     ; store input value from A6 in D1
        MOVE.B  D1,(A0)+    ; put it in address location
        BRA     MMBYTE      ; and loop!
MMWORD  ADDA.L  #16,A1  ; output will be 15 chars long + null
        MOVE.B  #0,-(A1)
        MOVE.B  #'?',-(A1)
        CLR.L   D1
        MOVE.W  (A0),D1
        BSR     HEX2MEM     ; writes memory content to -8(A1)
        ADDA.L  #4,A1       ; we only want 4 chars, not 8
        MOVE.B  #$9,-(A1)   ; a tabspace
        MOVE.L  A0,D1
        BSR     HEX2MEM     ; memory address to -8(A1)
        MOVE.B  #'$',-(A1)
        MOVE.B  #14,D0
        TRAP    #15         ; print
        MOVE.B  #2,D0
        TRAP    #15         ; read new value, if any
        CMPI.B  #0,(A1)
        BNE     MMWNEXT     ; skip memory address?
        ADDA.L  #2,A0       ; if yes, increment A0
        BRA     MMWORD      ; ...and loop
MMWNEXT CMPI.B  #'.',(A1)   ; else, check if done (entered '.')
        BEQ     MMDONE
        MOVEA.L A1,A6       ; new value to write in!
        BSR     MEM2HEX     ; store input value from A6 in D1
        MOVE.W  D1,(A0)+    ; put it in address location
        BRA     MMWORD      ; and loop!
MMLONG  ADDA.L  #20,A1  ; output will be 19 chars long + null
        MOVE.B  #0,-(A1)
        MOVE.B  #'?',-(A1)
        CLR.L   D1
        MOVE.L  (A0),D1
        BSR     HEX2MEM     ; writes memory content to -8(A1)
        MOVE.B  #$9,-(A1)   ; a tabspace
        MOVE.L  A0,D1
        BSR     HEX2MEM     ; memory address to -8(A1)
        MOVE.B  #'$',-(A1)
        MOVE.B  #14,D0
        TRAP    #15         ; print
        MOVE.B  #2,D0
        TRAP    #15         ; read new value, if any
        CMPI.B  #0,(A1)
        BNE     MMLNEXT     ; skip memory address?
        ADDA.L  #4,A0       ; if yes, increment A0
        BRA     MMLONG      ; ...and loop
MMLNEXT CMPI.B  #'.',(A1)   ; else, check if done (entered '.')
        BEQ     MMDONE
        MOVEA.L A1,A6       ; new value to write in!
        BSR     MEM2HEX     ; store input value from A6 in D1
        MOVE.L  D1,(A0)+    ; put it in address location
        BRA     MMLONG      ; and loop!
MMINV   BSR     INVALID
MMDONE  MOVEM.L (A7)+,D0-D1/A0-A1
        RTS

* MS -- store ascii (including null terminator) or hex in memory
MS      MOVEM.L D1/A1,-(A7)
        MOVE.B  (A6)+,D1    ; first '$'
        CMPI.B  #$24,D1     ; is it '$'?
        BNE     MSINV    ; wrong command usage
        BSR     MEM2HEX     ; D1 has 1st address in hex
        MOVEA.L D1,A1       ; store in A1
        MOVE.B  (A6)+,D1
        CMPI.B  #$20,D1     ; is it ' '?
        BNE     MSINV   ; wrong command usage
        MOVE.B  (A6)+,D1
        CMPI.B  #$24,D1     ; '$'?
        BEQ     MSHEX
```

```
        SUBA.L  #1,A6    ; have to put A6 back at start of ascii
MSASCII MOVE.B  (A6),(A1)+  ; put that char in (A1) and increment A1
        CMPI.B  #0,(A6)+    ; check if end and increment A6 to match A1
        BEQ     MSDONE  ; end of string
        BRA     MSASCII ; repeat
MSHEX   BSR     MEM2HEX ; hex number stored in D1
        CMPI.L  #$FF,D1 ; see size of number
        BLE     MSBYTE
        CMPI.L  #$FFFF,D1
        BLE     MSWORD
MSLONG  ADDA.L  #4,A1    ; move A1 to end of long word
        MOVE.B  D1,-(A1)     ; have to copy 4 bytes
        ROR.L   #8,D1        ; first one was copied, so look at next byte
        MOVE.B  D1,-(A1)     ; copy second byte
        ROR.L   #8,D1
        SUBA.L  #2,A1    ; done to counteract the next action
MSWORD  ADDA.L  #2,A1    ; move A1 to end of word
        MOVE.B  D1,-(A1)     ; will copy 2 bytes
        ROR.L   #8,D1    ; look at second one
        SUBA.L  #1,A1    ; to counteract the fact that MSBYTE doesn't predecrement
MSBYTE  MOVE.B  D1,(A1) ; copy one byte
        BRA     MSDONE
MSINV   BSR     INVALID
MSDONE  MOVEM.L (A7)+,D1/A1
        RTS

* BF -- fills block of memory with word pattern
BF      MOVEM.L D0-D3/D7/A1-A3,-(A7)
        MOVE.B  (A6)+,D1     ; first '$'
        CMPI.B  #$24,D1 ; is it '$'?
        BNE     BFINV ; wrong command usage
        BSR     MEM2HEX ; D1 has 1st address in hex
        MOVEA.L D1,A2    ;store in A2
        MOVE.B  (A6)+,D1     ; space in between addresses
        CMPI.B  #$20,D1 ; is it ' '?
        BNE     BFINV
        MOVE.B  (A6)+,D1     ; second '$'
        CMPI.B  #$24,D1
        BNE     BFINV
        BSR     MEM2HEX ; D1 has 2nd address in hex
        MOVEA.L D1,A3    ; both addresses have been read now
        CLR.L   D2       ; pattern will go in here
        MOVE.B  (A6)+,D1     ; space before the pattern
        CMPI.B  #$00,D1 ; no pattern given, use default
        BEQ     BFSTART
        CMPI.B  #$20,D1 ; is it ' '?
        BNE     BFINV
        MOVE.L  #3,D3    ; counter for remaining 3 digits (if there)
BFPATT  MOVE.B  (A6)+,D7     ; first byte of pattern
        TST.B   D7
        BEQ     BFSTART ; only one digit was given, use first one padded with a zero
        ASL.L   #4,D2    ; place first digit on the left part of the byte
        BSR     ASCII2NUM
        ADD.B   D7,D2    ; goes into the right part of the byte
        DBF     D3,BFPATT    ; debrease D3 and keep looping until all digits read
BFSTART MOVE.W  (A3),D3 ; TEST: if address2 not even, address error is raised
BFLOOP  CMPA.L  A2,A3
        BLE     BFDONE  ; done when A2 reaches A3
        MOVE.W  D2,(A2)+     ; write the pattern in memory. Address error raised if address1 not even
        BRA     BFLOOP
BFINV   BSR     INVALID
BFDONE  MOVEM.L (A7)+,D0-D3/D7/A1-A3
        RTS

* BMOV -- copies block of memory somewhere else
BMOV    MOVEM.L D1/A2-A4,-(A7)
        MOVE.B  (A6)+,D1     ; first '$'
        CMPI.B  #$24,D1 ; is it '$'?
        BNE     BMINV ; wrong command usage
        BSR     MEM2HEX ; D1 has 1st address in hex
        MOVEA.L D1,A2    ;store in A2
        MOVE.B  (A6)+,D1     ; space in between addresses
        CMPI.B  #$20,D1 ; is it ' '?
        BNE     BMINV
        MOVE.B  (A6)+,D1     ; second '$'
```

```
        CMPI.B  #$24,D1
        BNE     BMINV
        BSR     MEM2HEX ; D1 has 2nd address in hex
        MOVE.L  D1,A3   ; store in A3
        MOVE.B  (A6)+,D1    ; space in between addresses
        CMPI.B  #$20,D1 ; is it ' '?
        BNE     BMINV
        MOVE.B  (A6)+,D1    ; third '$'
        CMPI.B  #$24,D1
        BNE     BMINV
        BSR     MEM2HEX ; D1 has 3rd address in hex
        MOVE.L  D1,A4       ; store in A4
BMLOOP  CMPA.L  A2,A3
        BLE     BMDONE  ; done when A2 reaches A3
        MOVE.B  (A2)+,(A4)+ ; copy
        BRA     BMLOOP
BMINV   BSR     INVALID
BMDONE  MOVEM.L (A7)+,D1/A2-A4
        RTS

* BTST -- tests each bit (by setting and unsetting all) in a block of memory
BTERROR DC.B    'MEMORY ERROR FOUND AT LOCATION $00000000'
BTLOC   DC.B    $A,$D ; this and BTREAD point after for HEX2MEM to work
        DC.B    'Value expected: '
BTEXP   DC.B    '00',$A,$D
        DC.B    'Value read: 00'
BTREAD  DC.B    0
BTST    MOVEM.L D0-D1/A1-A3,-(A7)
        MOVE.B  (A6)+,D1    ; first '$'
        CMPI.B  #$24,D1 ; is it '$'?
        BNE     BTINV ; wrong command usage
        BSR     MEM2HEX ; D1 has 1st address in hex
        MOVEA.L D1,A2   ; store in A2
        MOVEA.L A2,A1   ; store copy for BTLOOP2
        MOVE.B  (A6)+,D1    ; space in between addresses
        CMPI.B  #$20,D1 ; is it ' '?
        BNE     BTINV
        MOVE.B  (A6)+,D1    ; second '$'
        CMPI.B  #$24,D1
        BNE     BTINV
        BSR     MEM2HEX ; D1 has 2nd address in hex
        MOVE.L  D1,A3   ; store in A3
        CLR.L   D1  ; needed to only look at bytes
BTLOOP1 CMPA.L  A2,A3   ; this loop tries bit pattern 1010
        BLE     BTPRELOOP2
        MOVE.B  #$AA,(A2)   ; write
        MOVE.B  (A2)+,D1    ; read
        CMPI.B  #$AA,D1     ; check correct
        BEQ     BTLOOP1     ; move to next byte
        LEA     BTREAD,A1   ; if here, there is a problem in memory!
        BSR     HEX2MEM_NOZ ; load everything to memory, to be able to print error
        LEA     BTEXP,A1
        MOVE.B  #'A',(A1)+
        MOVE.B  #'A',(A1)
        LEA     BTLOC,A1
        SUBA.L  #1,A2
        MOVE.L  A2,D1
        BSR     HEX2MEM
        LEA     BTERROR,A1
        MOVE.B  #13,D0
        TRAP    #15     ; print the error message
        BRA     BTDONE  ; stop execution
BTPRELOOP2  MOVEA.L A1,A2   ; copy was stored a while back to be able to start over
BTLOOP2 CMPA.L  A2,A3   ; this loop tries bit pattern 0101. Works the same as BTLOOP1
        BLE     BTDONE
        MOVE.B  #$55,(A2)   ; write
        MOVE.B  (A2)+,D1    ; read
        CMPI.B  #$55,D1     ; check correct
        BEQ     BTLOOP2     ; move to next byte
        LEA     BTREAD,A1   ; error in memory, act like before
        BSR     HEX2MEM_NOZ
        LEA     BTEXP,A1
        MOVE.B  #'5',(A1)+
        MOVE.B  #'5',(A1)
        LEA     BTLOC,A1
```

```
        SUBA.L  #1,A2
        MOVE.L  A2,D1
        BSR     HEX2MEM
        LEA     BTERROR,A1
        MOVE.B  #13,D0
        TRAP    #15
        BRA     BTDONE
BTINV   BSR     INVALID
BTDONE  MOVEM.L (A7)+,D0-D1/A1-A3
        RTS

* BSCH -- search for string literal in memory block
BSNO        DC.B    'Not found',0
BSYES       DC.B    'Found at location: $00000000'
BSYESADDR   DC.B 0
BSCH    MOVEM.L D1/A1,-(A7)
        LEA     BSNO,A1 ; will change if found
        MOVE.B  (A6)+,D1    ; first '$'
        CMPI.B  #'$',D1 ; is it '$'?
        BNE     BSINV   ; wrong command usage
        BSR     MEM2HEX ; D1 has 1st address in hex
        MOVEA.L D1,A2   ; store in A2
        MOVE.B  (A6)+,D1    ; space in between addresses
        CMPI.B  #' ',D1 ; is it ' '?
        BNE     BSINV
        MOVE.B  (A6)+,D1    ; second '$'
        CMPI.B  #'$',D1
        BNE     BSINV
        BSR     MEM2HEX ; D1 has 2nd address in hex
        MOVE.L  D1,A3   ; store in A3
        MOVE.B  (A6)+,D1    ; a space
        CMPI.B  #' ',D1
        BNE     BSINV
BSLOOP  CMPA.L A2,A3
        BEQ     BSDONE  ; stop if A2 reaches A3 (not found)
        MOVEA.L A6,A4   ; keep A6 for reference
        CMP.B   (A2)+,(A4)+ ; compare first char
        BNE     BSLOOP  ; look at next if different
        MOVE.L  A2,A5   ; keep A2 for reference
BSMAYB  CMPI.B  #0,(A4) ; see if we reached end of string
        BEQ     BSFOUND ; if we did, the whole string matched!
        CMP.B   (A5)+,(A4)+ ; else, compare next char
        BNE     BSLOOP  ; if not equal, have to check next possible word start
        BRA     BSMAYB  ; if equal, keep on looking in this word
BSINV   BSR     INVALID
        BRA     BSEND
BSFOUND MOVE.L  A2,D1   ; to tell where it was found
        SUBQ.L  #1,D1   ; was off by one
        LEA     BSYESADDR,A1
        BSR     HEX2MEM ; write address in the message
        LEA     BSYES,A1
BSDONE  MOVE.B  #13,D0
        TRAP    #15     ; print message: found or not found
BSEND   MOVEM.L (A7)+,D1/A1
        RTS

* GO -- executes another program
GO      MOVEM.L D0-D7/A0-A7,-(A7)   ; don't allow the program to change registers
        MOVE.B  (A6)+,D1    ; '$'
        CMPI.B  #$24,D1 ; is it '$'?
        BNE     GOINV   ; wrong command usage
        BSR     MEM2HEX ; D1 has address in hex
        MOVEA.L D1,A0   ;store in A0
        JSR     (A0)    ; execute the program
        BRA     GODONE
GOINV   BSR     INVALID
GODONE  MOVEM.L (A7)+,D0-D7/A0-A7
        RTS

* DF -- displays formatted registers
DF      MOVEM.L D0-D2/A0-A1,-(A7)
        LEA     STACK,A0
        ADDA.L  #4,A0   ; placed after A7 in stack
        LEA     DF_MSG_END,A1
DFLOOP  SUBQ.L  #1,A1   ; pass the $A at end of each line
```

```
        MOVE.L  #3,D2   ; number of registers per line - 1
DFLINE  MOVE.L  -(A0),D1    ; put register value in D1
        BSR     HEX2MEM     ; will store D1 in -8(A1)
        SUBQ.L  #4,A1   ; skip other characters
        DBF     D2,DFLINE   ; keep looping till line done
        CMP.L   #DF_MSG,A1
        BGT     DFLOOP
        ADDQ.L  #1,A1   ; put back at the front of the message
        MOVE.B  #13,D0
        TRAP    #15     ; print register value
        MOVEM.L (A7)+,D0-D2/A0-A1
        RTS

* EXIT -- terminates the program
EXIT    LEA     GOODBYE,A1
        MOVE.B  #13,D0
        TRAP    #15     ; print goodbye message
        ADDA.L  #4,A7   ; move past the PC stored in the stack
        ADDA.L  #MAX_IN_LEN,A7  ; move stack back to position prior to reading input
        MOVEM.L (A7)+,D0-D7/A0-A6   ; restore all registers in stack
        MOVEA.L STACK,A7
        BRA     END     ; exit program

* The 2 extra commands:
* BPRINT -- print as ascii a memory block
BPRINT  MOVEM.L D0-D1/A1-A3,-(A7)
        MOVE.B  (A6)+,D1    ; first '$'
        CMPI.B  #'$',D1 ; is it '$'?
        BNE     BPINV ; wrong command usage
        BSR     MEM2HEX ; D1 has 1st address in hex
        MOVEA.L D1,A2   ; store in A2
        MOVE.B  (A6)+,D1    ; space in between addresses
        CMPI.B  #0,D1   ; is it null?
        BEQ     BPNULL  ; read until null character found
        CMPI.B  #' ',D1 ; is it ' '?
        BNE     BPINV
        MOVE.B  (A6)+,D1    ; second '$'
        CMPI.B  #'$',D1
        BNE     BPINV
        BSR     MEM2HEX ; D1 has 2nd address in hex
        MOVE.L  D1,A3   ; store in A3
        MOVEA.L A6,A1   ; print from here
        MOVE.B  #0,1(A1)    ; make sure
        MOVE.B  #14,D0  ; for printing trap
BPBLOCK CMPA.L  A2,A3
        BLE     BPBDONE ; stop when A2 reaches A3
        MOVE.B  (A2)+,(A1)  ; put byte in (A1)
        TRAP    #15 ; print that byte!
        BRA     BPBLOCK
BPBDONE MOVE.B  #0,(A1)
        MOVE.B  #13,D0
        TRAP    #15     ; print a line feed and carriage return
        BRA     BPDONE
BPNULL  MOVEA.L A2,A1   ; no limit given, so print till null char found
        MOVE.B  #13,D0
        TRAP    #15     ; print!
        BRA     BPDONE
BPINV   BSR     INVALID
BPDONE  MOVEM.L (A7)+,D0-D1/A1-A3
        RTS

* CONV -- takes in hex and returns decimal, or viceversa
CONV    MOVEM.L D0-D1/A1,-(A7)
        MOVE.B  (A6)+,D1
        CMPI.B  #$24,D1 ; is it '$'?
        BEQ     CONVH2D ; if so, hex to dec
CONVD2H SUBQ.L  #1,A6   ; point back at first number
        BSR     MEM2DEC ; D1 contains the decimal number
        MOVEA.L A6,A1   ; number ready to print
        BSR     HEX2MEM_NOZ ; that number is written as hex in memory
        MOVE.B  #'$',-(A1)
        BRA     CONVDONE
CONVH2D BSR     MEM2HEX ; convert ascii to hex
        MOVEA.L A6,A1   ;number ready to print
        BSR     DEC2MEM ; convert it back to ascii but as decimal
```

```
CONVDONE MOVE.B #13,D0
        TRAP    #15 ; print result
        MOVEM.L (A7)+,D0-D1/A1
        RTS

*** HELPERS ***
* Print INVALID message:
INVALID MOVEM.L D0/A1,-(A7)
        LEA     INVALID_MSG,A1  ; command was invalid
        MOVE.B  #13,D0
        TRAP    #15     ; output invalid command
        MOVEM.L (A7)+,D0/A1
        RTS

* Takes X digits from (A6) in ascii and puts them in D1 as hex:
MEM2HEX MOVEM.L D0/D7,-(A7)    ; store in stack
        CLR.L   D1
        MOVE.B (A6)+,D7    ; read in next byte (prime read)
        CMPI.B  #$30,D7
        BLT     M2HDONE ; reached some whitespace or non-numeric ascii
M2HNEXT BSR     ASCII2NUM   ; byte to hex digit, in D7
        ADD.B   D7,D1
        MOVE.B (A6)+,D7    ; read in next byte (prime read)
        CMPI.B  #$30,D7
        BLT     M2HDONE ; reached some whitespace or non-numeric ascii
        ASL.L   #4,D1   ; skip this the last time
        BRA     M2HNEXT ; loop again because not done
M2HDONE SUBA.L #1,A6   ; leave A6 pointing at byte immediately after last number
        MOVEM.L (A7)+,D0/D7    ; restore from stack
        RTS

* Takes byte in ascii in D7 and converts it to digit in D7:
* Assumes 0-9 or A-F
ASCII2NUM   CMPI.B #$40,D7
        BLT A2NSKIPPY
        SUBQ.B  #$7,D7   ; only for A-F
A2NSKIPPY   SUB.B   #$30,D7
        RTS

* Takes 8 digits from D1 in hex and puts them into -8(A1) in ascii:
HEX2MEM MOVEM.L D0/D2/D7,-(A7)    ; store in stack
        CLR.L   D0   ; counter
H2MNEXT MOVE.L  D1,D7
        MOVE.L D0,D2
H2MRIGHT    SUBQ.W  #1,D2
        BLT     H2MDONE
        LSR.L   #4,D7   ; that upper byte to lowest by -> only one left
        BRA     H2MRIGHT
H2MDONE BSR     NUM2ASCII   ; convert to ascii in D7
        MOVE.B  D7,-(A1)
        ADDQ.W  #1,D0
        CMPI.W  #8,D0
        BLT     H2MNEXT
        MOVEM.L (A7)+,D0/D2/D7
        RTS

* Takes X digits from D1 in hex and puts them into -X(A1) in ascii (no trailing zeros):
HEX2MEM_NOZ MOVEM.L D0/D2/D7,-(A7)    ; store in stack
        CLR.L   D0   ; counter
H2MZNEXT MOVE.L  D1,D7
        MOVE.L D0,D2
H2MZRIGHT    SUBQ.W  #1,D2
        BLT     H2MZDONE
        LSR.L   #4,D7   ; that upper byte to lowest by -> only one left
        BRA     H2MZRIGHT
H2MZDONE TST.L   D7
        BEQ     H2MZEND      ; if number done
        BSR     NUM2ASCII   ; convert to ascii in D7
        MOVE.B  D7,-(A1)
        ADDQ.W  #1,D0
        CMPI.W  #8,D0
        BLT     H2MZNEXT
H2MZEND  MOVEM.L (A7)+,D0/D2/D7
        RTS
```

```
* Takes digit in D7 and converts it to ascii byte in D7:
* Assumes 0-9 or A-F
NUM2ASCII   AND.L  #$0F,D7 ; mask and take only smallest hex digit
        CMPI.B  #$A,D7
        BLT N2ASKIPPY
        ADDQ.B  #$7,D7   ; only for A-F
N2ASKIPPY   ADD.B   #$30,D7
        RTS

* Takes X digits from (A6) in ascii and puts them in D1 as dec:
MEM2DEC MOVEM.L D0/D7,-(A7)    ; store in stack
        CLR.L   D1
        MOVE.B (A6)+,D7    ; read in next byte (prime read)
        CMPI.B  #$30,D7
        BLT     M2DDONE ; reached some whitespace or non-numeric ascii
M2DNEXT BSR     ASCII2NUM   ; byte to hex digit, in D7
        ADD.B   D7,D1
        MOVE.B (A6)+,D7    ; read in next byte (prime read)
        CMPI.B  #$30,D7
        BLT     M2DDONE ; reached some whitespace or non-numeric ascii
        MULU    #10,D1   ; skip this the last time
        BRA     M2DNEXT ; loop again because not done
M2DDONE  SUBA.L #1,A6   ; leave A6 pointing at byte immediately after last number
        MOVEM.L (A7)+,D0/D7    ; restore from stack
        RTS

* Takes number from D1 in dec and puts them into -X(A1) in ascii:
DEC2MEM MOVEM.L D2/D7,-(A7)    ; store in stack
        MOVE.L  D1,D2
D2MLOOP DIVU    #10,D2
        MOVE.L  D2,D7
        SWAP.W  D7
        BSR     NUM2ASCII
        MOVE.B  D7,-(A1)
        AND.L   #$0000FFFF,D2 ; make sure we use only word in next divisions
        TST.W   D2
        BNE     D2MLOOP
        MOVEM.L (A7)+,D2/D7
        RTS

*** EXCEPTION HANDLERS ***
ADDRERR MOVEM.L D0/A1,-(A7)
        MOVEM.L D1/A0,-(A7) ; specific for this interrupt
        LEA     ADDRERR_MSG,A1
        MOVE.B  #13,D0
        TRAP    #15
        BRA     INTERR_REG  ; print the special registers
BERR    MOVEM.L D0/A1,-(A7)
        MOVEM.L D1/A0,-(A7) ; specific for this interrupt
        LEA     BERR_MSG,A1
        MOVE.B  #13,D0
        TRAP    #15
        BRA     INTERR_REG  ; print the special registers
ILLINS  MOVEM.L D0/A1,-(A7)
        LEA     ILLINS_MSG,A1
        BRA     INTERR
PRIVERR MOVEM.L D0/A1,-(A7)
        LEA     PRIVERR_MSG,A1
        BRA     INTERR
DIV0    MOVEM.L D0/A1,-(A7)
        LEA     DIV0_MSG,A1
        BRA     INTERR
CHKERR  MOVEM.L D0/A1,-(A7)
        LEA     CHKERR_MSG,A1
        BRA     INTERR
LINEA   MOVEM.L D0/A1,-(A7)
        LEA     LINEA_MSG,A1
        BRA     INTERR
LINEF   MOVEM.L D0/A1,-(A7)
        LEA     LINEF_MSG,A1
        BRA     INTERR
INTERR_REG  ; only BERR and ADDRERR do this
        MOVEA.L A7,A0
        ADDA.L  #24,A0  ; A0 is pointing right below SSW, BA and IR
        MOVEA.L #STACK,A1
```

```
        SUBA.L  #60,A1  ; write message in the input space of the stack (currently unused)
        MOVE.B  #0,-(A1)    ; null terminator
        CLR.L   D1
        MOVE.W  -(A0),D1    ; SSW in D1
        BSR     HEX2MEM
        ADDQ.L  #4,A1       ; only want SSW to be a word
        MOVE.B  #' ',-(A1)
        MOVE.L  -(A0),D1    ; BA in D1
        BSR     HEX2MEM
        MOVE.B  #' ',-(A1)
        CLR.L   D1
        MOVE.W  -(A0),D1    ; IR in D1
        BSR     HEX2MEM
        ADDQ.L  #4,A1       ; only want IR to be a word
        MOVEM.L (A7)+,D1/A0 ; restore these specific registers
INTERR  MOVE.B  #13,D0
        TRAP    #15 ; print corresponding message for that interrupt
        BSR     DF  ; print registers
        MOVEM.L (A7)+,D0/A1 ; do here to be able to modify values of A7
        LEA     STACK,A7    ; next 3 instructions put A7 at beginning of input space in stack
        SUBA.L  #60,A7  ; 15 registers that occupy 4 bytes each (2*4 = 8 bits)
        SUBA.L  #MAX_IN_LEN,A7  ; the input space
        BRA     PROMPT

*** PROGRAM FOR TESTING GO ***
    ORG $3200
        MOVEA.L #$4020,A1
        MOVE.L  #$48492100,(A1)
        MOVE.B  #13,D0
        TRAP    #15     ; print secret message
        RTS

END
    END     START       ; last line of source
```