MUN-88 MONITOR 1.0

Reference Manual

by

David Skoll

Joseph Liang

October 1988

Reformatted by:

Charles Robertson, June 2000

Table of Contents

1. Introduction

The MUN-88 Monitor is a program which controls the operation of the MUN-88 Single-board Computer. It is written in 8088 assembly language, and is contained in a 2764A EPROM on the MUN-88 Single-board Computer. A different version of the monitor (the monitor simulator) is available which runs on the IBM PC, and which simulates (as much as possible) the MUN-88 environment on a PC. This simplifies users' software development, since much of their software can be developed and tested directly on the IBM PC rather than having to download it to the MUN-88 each time it must be run.

The monitor allows the user to examine and modify the computer's memory, to examine and modify the CPU registers, and to test and run programs on the MUN-88. In addition, it contains many useful routines which can be used by user-written programs. Finally, the command set of the monitor is expandable - expansion ROMs can place new commands into the structure of the monitor with a minimum of difficulty. Thus, the capability of the software can be extended, and the computer can be programmed for a myriad of uses.

This manual is divided into 6 sections. This section, the introduction, explains what the monitor is. Section 2 provides some more detailed technical background about the monitor, and describes how to get started. Section 3 describes the monitor commands and how to use them. Section 4 describes monitor function calls available to the user. And section 5 explains the mechanism for enhancing the monitor's command set with the use of an expansion ROM. Finally, section 6 describes differences between the monitor on the MUN-88 Single-board Computer and the monitor simulator which runs on the IBM PC.

2. Getting Started

To start the monitor, connect the MUN-88 to its power supply. Load a communications package into an IBM PC or compatible, and connect the MUN-88 serial port to the PC serial port. Set the communications parameters as follows:

Speed:	See below
Data bits:	8
Stop bits:	1
Parity:	None
Duplex:	Full

To set the communication speed of the MUN-88, adjust the DIP switches as shown in the following table. (The speed is shown in baud.) The switches are numbered from 1 to 8, with 1 being on the left and 8 on the right (if you hold the computer with the reset switch at the bottom.) For each switch, OFF is down and ON is up.

Speed	***Switch 1***	***Switch 2***	***Switch 3***
75	ON	ON	ON
110	OFF	ON	ON
150	ON	OFF	ON
300	OFF	OFF	ON
600	ON	ON	OFF
1200	OFF	ON	OFF
2400	ON	OFF	OFF
4800	OFF	OFF	OFF

Turn on the MUN-88 Single-board Computer's power supply. The LEDs on the MUN-88 should cycle through a self-test pattern, and then go dark. At this point, your screen should look like this:

```
MUN-88 Monitor 1.0
Copyright (C) 1988 by David Skoll and Joseph Liang.
Free memory begins at 0032:0000


.
```

The cursor should be flashing just after the period. The period is called the **dot prompt** (like in dBASE) and indicates that the MUN-88 is ready to accept a command. None of the LEDs should be on once the dot appears. If instead of this, your screen remains blank and the LED display indicates an alternating on-and-off pattern, then it means that the RAM chip on the MUN-88 is bad.

The "Free memory" message indicates the lowest address at which you should load programs or modify memory. All memory below this location (ie, from 0000:0000 to 0000:031F) is used by the monitor system and should not be modified. If you do modify memory below this region, the monitor could crash.

2.1. Register Images

The MUN-88 Monitor maintains images of the CPU registers. These images are placed into the registers just before executing a user's program. When control is passed back to the monitor, the values in the CPU registers are placed into the images before being modified. Thus, as far as the user is concerned, the images represent the values of the CPU registers just after the last user instruction was executed. Whenever you examine or modify registers, you actually modify the register images. Thus, you can change any of the registers without fear of crashing the monitor. However, you may crash a user-written program if you change the register values carelessly.

When the monitor starts up, the register images contain the following values:

- CS:IP point to the first instruction in the monitor ROM. Thus, CS=FE00 and IP=0000

- SS:SP point to the top of the user stack space. The monitor reserves 64 words (128 bytes) of stack space for the user. If you need more, you must set up your own stack.

- ES:0000 points to the first free memory location. Thus, ES contains the value 0032 printed in the "Free memory" message.

- DS points to the monitor's data segment. This is 0000.

- All other registers contain 0000, and all flags are cleared.

2.2. Alarms

The monitor maintains a set of eight flags called alarms. Whenever an error or unexpected event occurs, one of these flags is set. This is called "raising an alarm." In addition, an LED corresponding to the alarm is lit to indicate to the user that an alarm has been raised. (The display of alarms may be deactivated, but alarms are still maintained internally.) The LEDs are numbered from 0 to 7, with 0 on the left and 7 on the right. The alarms are:

0 -Software error - this alarm is never raised by the monitor, but is available for user-written programs to raise.

1 - Transmit timeout - the USART timed out when the CPU attempted to transmit. This indicates a problem with the USART or the baud rate generator.

2 - Receive error - a framing or overrun error was detected. It could be that your communications package is transmitting at the wrong speed, or there could be a problem with the USART, the RS-232 receivers or the baud rate generator.

3 - Input buffer overflow - the monitor maintains a 40-character "typeahead" buffer. If you type very quickly (or type a lot while the monitor is busy executing a command) the buffer will overflow. Extra characters are discarded.

4 - Long line truncated - if you try to type a command line longer than 80 characters, this alarm will be raised.

5 - Illegal function call - raised by the monitor if a user-written program makes an illegal function call. See section 4 for more details.

6 - Bad hook - this is never raised by the monitor, but may only be raised by an expansion ROM if it is present. It indicates a software error in the interface between the expansion ROM software and the monitor.

7 - Unknown/unexpected interrupt received - this may occur if a user-written program makes a software interrupt to an undefined vector, or if additional devices are connected to the 8259A interrupt controller without having interrupt service routines provided for them.

3. Monitor Commands

3.1. Syntax Conventions

Each monitor command is a string of letters. Letters shown in upper-case must be typed; letters in lower-case are optional. (The actual letters may be entered in upper- or lower-case.) For example, the syntax of the QALM command is:

QAlm

This means that you can type qalm, QAL, Qa. etc. You must type at least QA, since the first two letters are upper-case.

When typing a command, you can use the backspace key to correct errors. Press ENTER when you have finished typing the command. If you press ESC, anything you have typed is cancelled and you start out from the dot prompt with a clean line.

3.2. Parameters

Some commands require parameters. These parameters may be optional; if they are, this is indicated by angle braces. The various types of parameters are:

seg:off - this is a segment:offset address consisting of two unsigned hexadecimal numbers separated by a colon. There may not be embedded blanks anywhere within the parameter. Letters may be upper- or lower-case. Examples of a **seg:off** parameter are:

0000:0000

09ab:9a9a

0:0

feda:88

value - this is an unsigned hexadecimal number from 0 to FFFF. (In some contexts, the range may be limited to 0 to FF.) Examples of a **value** parameter are:

34

fe90

9a8

count - this parameter is the same as a **value** parameter, except that the value 0 is prohibited.

port - this is an unsigned hexadecimal number from 0 to FFFF denoting an I/O port address.

register - this parameter is the name of a 16-bit 8088 register. Valid names are:

AX, BX, CX, DX, SI, DI, SP, BP, CS, DS, ES, SS, IP and FL (FL stands for FLAGS)

Note that the name of a register may be used wherever a 16-bit **value**, **port** or **count** parameter is required, or in either component of a **seg:off** parameter. If this is done, then the monitor uses the contents of the specified register image as the value. Thus, the following parameters are valid:

```
ES:0000  (seg:off type)
095:IP  (seg:off type)
DX        (port, count or value type)
```

number - this parameter must be a number from 0 to 7

ON|OFF - this parameter must be either the word "ON" or the word "OFF".

3.3. Monitor Commands

DISPLAY COMMAND

Syntax: Display <seg:off <count>>

Examples:

```
D 0:0
DI CS:00 10
display 98:di
d
```

Operation: Displays the contents of memory starting from seg:off and continuing for count bytes. If count is omitted, then a default value of 128 is used. If both parameters are omitted, then 128 bytes of memory are displayed starting from where the last display command left off. If there was no last display command, then memory is displayed starting from 0000:0000. However, if a program has been downloaded using the DLOAD command, then a DISPLAY command without parameters will display memory starting from the load address of the program.

The display command places a '-' character between bytes of memory aligned on half-paragraph boundaries (i.e., every 8 bytes). It starts a new line for bytes aligned on paragraph boundaries.

MODIFY COMMAND

Syntax: Modify seg:off <value>

Examples:

```
M 0000:0454 7F
mod ds:di
modify 9:2
```

Operation: Places value into the location seg:off. Note that value can range from 00 to FF, and cannot be the name of a CPU register. If value is omitted, you will be placed in the interactive modify mode. In this mode:

- Type a value and press ENTER to modify a memory location and move to the next location.

- Press ENTER without typing a value to leave the contents of a memory location unchanged and move to the next location.

- Press ESC to return to the normal command prompt.

DREG COMMAND

Syntax: DReg <register>

Examples:	DREG
	dr fl
	dr ax

Operation: Displays the contents of the specified register. If register is not supplied, then the command displays the contents of all the registers.

REG COMMAND

Syntax: Reg <register <value>>

Examples:	R
	reg ax
	r cx 5098
	r ds cs

Operation: Places value into the specified register. If value is not specified, it prompts for a value to place in the register. If neither register nor value is specified, then the command goes into an interactive mode (similar to the MODIFY command) which allows you to selectively modify all the registers. As with the modify command, type a value and press ENTER to change a register. Simply press ENTER without typing anything to leave a register unchanged. And press ESC to return to the dot prompt.

Note that if the FL register is modified interactively, you are prompted for each flag. Enter a 1 or a 0 for each flag you want to change. If you don't want to change a particular flag, just press ENTER without typing anything else.

SINGLE COMMAND

Syntax: Single <ON|OFF>

Examples:	single
	S ON
	sing off

Operation: The monitor is capable of executing user-written programs one instruction at a time, and then pausing to display registers. This mode is selected with the SINGLE ON command. To return to the normal mode, type SINGLE OFF. Typing SINGLE with no parameters causes the monitor to display the current setting of the single-step feature. See the GO command for more information.

BRK COMMAND

Syntax: Brk seg:off

Examples:	BRK 0:57
	b cs:0347
	br 567A:ax

Operation: This command sets a breakpoint at the specified address. Breakpoints can be set only in RAM, and should be set at the first byte of an 8088 opcode. Up to 8 breakpoints may be active at once. When the monitor is executing a user-written program and it encounters a breakpoint, it halts execution and displays the contents of the registers. The instruction occupying the breakpoint address is not executed - the instruction just prior to it is. See the GO command for more information.

NOBRK COMMAND

Syntax: Nobrk <number>

Examples:	NOBRK
	n 6
	No 2

Operation: This program clears the specified breakpoint. The breakpoints are numbered from 0 to 7, and whenever the BRK command is used to set a breakpoint, a message indicating the breakpoint's number is displayed. If the number parameter is omitted, all breakpoints are cleared.

QBRK COMMAND

Syntax: QBrk

Examples:
```
QBRK
qb
```

Operation: This command displays the status of each of the 8 available breakpoints.

COPY COMMAND

Syntax: Copy seg:off1 seg:off2 count

Examples:
```
COPY 9451:0947 9985:0090 100
c cs:0 ds:0 cx
c 0:0 es:F000 A0
```

Operation: This copies bytes starting at seg:off1 to memory starting at seg:off2. It copies a total of count bytes. Note that seg:off1 can be higher or lower than seg:off2, and the two areas of memory can overlap. However, the COPY command will not copy across segment boundaries - for example, the command COPY 0000:0000 0090:FFF0 20 will cause a wraparound from 0090:FFFF to 0090:0000 as the sixteenth byte is copied. Note you can "trick" COPY by exploiting the partial decoding of the MUN-88 Computer so that copy "thinks" the source is higher than the destination while it actually may be lower because the same physical memory is repeated in the memory map. If you do this, it could cause the COPY command to give erroneous results.

IN COMMAND

Syntax: In port

Examples:
```
IN 30
i dx
```

Operation: This command reads the specified input port and displays the byte read. See the hardware description of the MUN-88 Computer for an I/O map.

OUT COMMAND

Syntax: Out port value

Examples:	OUT 30 0
	o dx F4

Operation: This command sends the specified 1-byte value to the output port. See the hardware description of the MUN-88 Computer for an I/O map.

CLRALM COMMAND

Syntax: CLralm

Examples:	CL
	clralm

Operation: This command clears all alarms which have been raised.

QALM COMMAND

Syntax: QAlm

Examples:	QA
	qalm

Operation: This command queries the alarm status and displays a list of alarms which are currently raised.

ALM COMMAND

Syntax: Alm <ON|OFF>

Examples: ALM

a on

AL off

Operation: This command sets or queries the alarm annunciator status. If you use the ALM ON command, then all alarms which are raised are displayed on the LED display. This is the setting when the monitor first starts. ALM OFF disables the display of alarms on the LEDs. The alarm flags are still maintained internally (and may be queried with the QALM command), but the LED display is freed so that user-written software can use it for any purpose it desires. If you type ALM with no parameters, the current setting of the LED annunciators is displayed.

LED COMMAND

Syntax: Led value

Examples: LED 55

l ff

le 0

Operation: This command controls the LED display. All of the LEDs corresponding to "1" bits of the one-byte value parameter are lit, and all LEDs corresponding to "0" bits are extinguished. While this can be accomplished using the OUT command (OUT 30 xx), the LED command has two differences: First, it updates the RAM image of the LED display so that when alarms are raised and lowered, the monitor software operates the display correctly. Secondly, the sense of the bits is inverted - if you use the OUT command, a "0" bit will light an LED and a "1" bit will extinguish it.

HELP COMMAND

Syntax: Help

Examples: help

H

Operation: This command displays a list of all monitor commands along with a brief description of how to use them and what they do.

GO COMMAND

Syntax: Go <seg:off>

```
Examples:     GO 0000:0345
              g
              g es:f00
```

Operation: This command begins execution of a user program at the specified address. If no parameter is supplied, execution begins at CS:IP, where CS is the current CS register image and IP is the current IP register image. Thus, the command GO is equivalent to GO CS:IP.

If the single-step mode is on, then the GO command executes one instruction, displays the contents of the registers and the instruction about to be executed, and then returns to the command prompt. To continue executing successive instructions, keep typing the GO command with no parameters. If a breakpoint is set at an instruction, but the single-step mode is on, then the single-step interrupt will override the breakpoint.

If single-step mode is off, the program will execute until a breakpoint is encountered, or until a program terminate interrupt (INT 6) instruction is executed. If a breakpoint is encountered, the contents of the registers and the instruction at which the breakpoint was set are displayed.

Note that if the GO command starts execution at a breakpoint, the monitor behaves slightly differently. It disables the breakpoint, executes one instruction, then enables the breakpoint and continues execution. This is done so that if a breakpoint is encountered, the next GO command will continue program execution from where it left off. For example, consider the following program:

```
start:   mov     cx, 8
                 moval, 0FEH
do_it:   out     30H, al
                 rol   al, 1
                 calldelay_10ms
                 loopdo_it
                 int6      ; program terminate interrupt
```

(Assume that DELAY_10MS is a subroutine which produces a delay of 10 ms without modifying any registers.) Suppose you place a breakpoint at the OUT 30H, AL instruction. If you begin execution at START, the breakpoint will be encountered, and CS:IP will point to the OUT instruction. Also, CX will contain 0008, and AL will contain FE. If you then type GO, the OUT instruction will be executed, followed by the rotate, the CALL and the LOOP. Then, the breakpoint will be encountered, and CX will contain 0007 and AL will contain FD. Thus, you can step through the loop one iteration at a time by repeatedly typing GO, because of the automatic disabling and re-enabling of the breakpoint at the GO address. If the breakpoint were not disabled, then each time you typed GO, you would simply encounter the same breakpoint and the loop would never be executed.

Note: Because of the architecture of the 8088, certain instructions cannot be single-stepped. Any instruction which moves a value into a segment register will not be single-stepped. Instead, it and the instruction following it will be executed as an atomic instruction. For example, the following instructions will be single-stepped as a single unit:

```
MOVSS, AX
MOVSP, DX
```

Note: You cannot set breakpoints in ROM, but you can single-step ROM instructions. However, DO NOT single-step any of the monitor code (especially the single-step or breakpoint interrupt service routines), since this could cause stack conflicts which can crash the system. To use single-stepping or breakpoints, there must be at least 6 words (12 bytes) of stack space available on the user stack.

Note: The single-step command overrides the value in the Trace Flag. Thus, changing TF from the monitor has no effect. Also, your program should NOT change the value of TF.

DLOAD COMMAND

Syntax: DLoad seg:off

Examples:

```
DLOAD 34:0000
DL ES:0
dl 1a0:ip
```

Operation: This command downloads a binary image from the IBM PC into memory beginning at the specified segment:offset. To create a user-written file to download, follow these steps:

1) Write your assembler program. Make sure that all data and code is contained in one segment. Start the program from offset 0. For example:

```
myseg segment
            assume      cs:myseg, ds:myseg, es:myseg

; Set up the segment registers

start:      mov         ax, cs
            mov         es, ax
            mov         ds, ax

; Place your code here
```

```
            ; ; ;       ETC         ; ; ;

            ; End of program
                        int         6

            data1       db          ?
            string      db          "This is a string"
            ; Place more data declarations here
            ; ; ;       ETC         ; ; ;

            myseg ends
                        end         start
```

Note: When setting up segment registers to point to your data area, DO NOT use something like MOV AX, MYSEG and then MOV DS, AX. If you use this method, then when you convert your file to a binary image, you will be asked to supply segment fixups. This will imply that you know exactly where your program will reside in memory, and the resulting binary image will not be relocatable. Instead, use the method illustrated above - use the value in CS to set up the other segment registers to point to your segment.

2) Assemble your file

3) Link your file. Ignore the warning about the stack segment.

4) Run EXE2BIN on the EXE file produced by LINK. This will produce a binary image of your program.

To download the file to the MUN-88, use the DLOAD command to specify the segment and offset. (The offset should be 0.) Once the message "Ready to download..." appears, run the program MUN88DL.EXE to transmit the file. This program is run from DOS by typing MUN88DL filename. When the program has finished, return to your communications package and press ENTER to send a carriage return to the MUN-88. You should see a message indicating whether or not the download succeeded. If the MUN-88 appears to have hung up, send a ^C (ETX) followed by three carriage returns to regain control. Note that the DLOAD command lowers the receive error alarm before downloading; if this alarm is raised when the command has finished, then you know that a communications error occurred during the file transmission.

DOWNLOAD PROTOCOL:

The protocol for the DLOAD command is very simple. Once the command is issued, the monitor keeps receiving characters from the terminal. It discards all characters until a STX (^B) character is received. It then begins storing characters in memory until an ETX (^C) is received. If an ETX occurs as part of the binary image, the terminal should send the sequence ESC ETX. If an ESC is part of the binary image, then the sequence ESC ESC should be sent. After the ETX signifying the end of file has been sent, two bytes containing a checksum should be sent, with the low-order byte sent first. The checksum is the (16-bit) sum of all characters sent, NOT INCLUDING the first STX, the last ETX, or the extra ESC characters sent. (ie, it is summed over all characters actually in the binary image.) The monitor then sits in a loop looking for a carriage return. Once it receives a carriage return, it assumes that the communications package is back up. It then displays a message indicating the success or failure of the download, and returns to the command prompt.

4. Monitor Function Calls

The MUN-88 monitor allows users to request 26 function calls from user-written programs. To use a function call, load the number of the function call into CH. Set up any other necessary registers, and then execute an INT 7 instruction. You can assume that no registers are changed by the function call except those specifically mentioned in each function call's description.

The description of each function call is in the following format:

Name:	The name of the function call
Number:	The number in DECIMAL which should be loaded into CH to perform the function call.
Operation:	A description of what the function call does.

4.1. Function Call Descriptions

Name: FN_GETC

Number: 0

Operation: This function gets a character from the terminal input buffer. If no character is in the buffer, it waits until a character is sent from the terminal and returns with the character. The character returned is placed in the AL register.

Name: FN_GETC_NOWAIT

Number: 1

Operation: This function places a character from the terminal input buffer into AL if one is available, and clears the carry flag. If no character is available in the buffer, then carry is set and AL is undefined. Note that the USART interrupt should be enabled and IF should be set for the duration of the program for this function to work correctly, since input from the terminal is interrupt-driven.

Name: FN_GETLIN

Number: 2

Operation: This function gets a line of input from the terminal. It places the characters in the buffer pointed to by ES:DI. CL should contain the maximum number of characters to accept. If more characters are typed, they are ignored and alarm #4 is raised. Upon return, the line is terminated with a zero byte. If RETURN was pressed, then carry is cleared. If ESC was pressed, carry is set. No carriage return/linefeed is printed after the line was entered.

Note: If you want to input a line of at most n characters, you should place n into CL, but reserve a buffer of (n+1) characters. This is necessary to accommodate the zero byte which terminates the line, and is similar to the way that C handles strings.

Name: FN_PUTC

Number: 3

Operation: This function transmits the character in AL to the terminal.

Name: FN_PUTLIN

Number: 4

Operation: This function sends a series of characters to the terminal. The characters are transmitted starting from DS:BX until a zero-byte terminator is encountered. The zero byte itself is not sent.

Name: FN_NICELIN

Number: 5

Operation: This function prepares a line for easy parsing. Note that all of the monitor parsing functions such as FN_GETSEG_OFF, FN_GETVAL_CHK, etc. assume that they are being passed a pointer to a zero-terminated line that has been processed by FN_NICELIN. This function takes the zero-terminated character string pointed to by DS:SI and performs the following operations:

- Leading and trailing spaces are deleted

- Multiple spaces are compressed into single spaces

- All letters are converted to upper-case

Name: FN_LED

Number: 6

Operation: This function takes the value in AL and turns on LEDs which correspond to "1" bits, and turns off LEDs which correspond to "0" bits. It also updates the internal RAM image of the LED display maintained by the monitor.

Name: FN_LED_ON

Number: 7

Operation: This function turns on LEDs which correspond to "1" bits in AL, and updates the internal RAM image of the LED display.

Name: FN_LED_OFF

Number: 8

Operation: This function turns off LEDs which correspond to "1" bits in AL, and updates the internal RAM image of the LED display.

Name: FN_QALM

Number: 9

Operation: This function returns the alarm status in AL. If a bit is set, then the corresponding alarm is active.

Name: FN_CLRALM

Number: 10

Operation: This function clears all alarms.

Name: FN_RAISE_ALARM

Number: 11

Operation: This function raises the alarm whose number is specified in the 3 low-order bits of AL. For example, if AL contains 05, then alarm 5 is raised.

Name: FN_FLUSH

Number: 12

Operation: This function empties the internal typeahead buffer. Any characters in the buffer are discarded.

Name: FN_GETNUM

Number: 13

Operation: This function parses a string of ASCII characters pointed to by DS:SI. If a sequence of valid hexadecimal characters are found, then upon exit, DX contains the number, and DS:SI point to the first non-hexadecimal character found. Carry is cleared. If no valid hexadecimal characters are found, then DX and SI are unchanged and carry flag is set.

Name: FN_PUTNUM_BYTE

Number: 14

Operation: This function converts the value in DH into two ASCII characters showing the hexadecimal representation of the value. The characters are placed in memory into the area pointed to by ES:DI. Upon exit, the contents of DI are two more than upon entry.

Name: FN_PUTNUM_WORD

Number: 15

Operation: This function converts the value in DX into four ASCII characters showing the hexadecimal representation of the value. The characters are placed in memory into the area pointed to by ES:DI. Upon exit, the contents of DI are four more than upon entry.

Name: FN_PUTSEG_OFF

Number: 16

Operation: This function converts the values in AX:DX into nine ASCII characters showing the hexadecimal representation of a segment:offset. The first four characters are the value in AX, the fifth character is a colon, and the last four characters are the value in DX. The characters are placed in memory beginning at ES:DI. Upon exit, the contents of DI are 9 more than upon entry.

Name: FN_ALM

Number: 17

Operation: This function sets or queries the status of the LED annunciators. If AL contains 0, then the alarm display is disabled. If AL contains FF, then the alarm display is enabled. If AL contains 1, then upon exit, AL is set to 0 or non-zero, depending on the current status of the alarm display. (0 indicates that it is disabled.) If AL contains a value other than 0, 1, or FF, then behaviour is undefined.

Name: FN_ERROR

Number: 18

Operation: This function is an error exit path. It resets the stack segment and stack pointer to the top of system stack. It resets ES and DS to point to the system data segment, and then jumps back to the main loop of the monitor (command prompt.) This function call is probably most useful for expansion ROM programs rather than user-written programs.

Name: FN_GETVAL

Number: 19

Operation: This function is very similar to FN_GETNUM, except that it allows a valid register name to be specified in place of a hexadecimal constant. If a valid register name is found, then DX contains the value in the register's image. Otherwise, the line is parsed for a hexadecimal constant. Parsing begins at DS:SI. Upon exit, carry is set if no valid value was found; otherwise, carry is cleared. SI points to the first non-hexadecimal character found if the parameter was a hexadecimal constant, or the first character after the register name if the parameter was a register name.

Name: FN_GETSEG_OFF

Number: 20

Operation: This function parses the line pointed to by DS:SI for a seg:off type parameter. Either or both components may be a valid register name. It is also verified that the first non-valid character found is either a space or a zero byte. If a valid parameter is found, then AX contains the segment component, DX contains the offset component, and SI points to the space or zero byte after the parameter. If a valid parameter is not found, then an error message is printed and the FN_ERROR function is invoked. Thus, execution of your program will be terminated by an invalid parameter. Make sure that's what you want!

Name: FN_GETVAL_CHK

Number: 21

Operation: This function is identical to FN_GETVAL, except that it verifies that a space or zero-byte follows the parameter, and if an invalid parameter is found, then a message is printed and FN_ERROR is invoked. Thus, execution of your program will be terminated by an invalid parameter. Make sure that's what you want!

Name: FN_CRLF

Number: 22

Operation: This function sends a carriage return followed by a linefeed to the terminal.

Name: FN_COND_CRLF

Number: 23

Operation: If the current cursor position is not at the left-most column, then a carriage return/linefeed sequence is sent. Otherwise, no action is taken.

Name: FN_TAB

Number: 24

Operation: This function sends the appropriate number of spaces to the terminal such that the cursor is moved to the column whose number is in CL. Columns are numbered (from left to right) from 0 to 79. If the current position of the cursor is greater than or equal to that specified in CL, no action is taken.

Name: FN_LOWER_ALARM

Number: 25

Operation: This is the complement of FN_RAISE_ALARM. It lowers the alarm whose number is specified in the 3 low-order bits of AL. For example, if AL contains 03, then alarm 3 is lowered.

4.2. Program Terminate Interrupt

A user-written program which is started with the GO command should execute an INT 6 instruction when it has finished executing. This returns control to the monitor.

5. The Expansion ROM

The capabilities of the MUN-88 may be enhanced by mapping an 8K EPROM from FC00:0000 to FC00:1FFF. The code in this EPROM can be executed automatically and can thread itself into the monitor.

5.1. *Detection of the Expansion ROM*

The expansion ROM is detected by checking for the nine-character sequence "EXPANSION" in the bytes from FC00:0000 to FC00:0008. This is called the signature sequence. If these bytes are found, then control is transferred to FC00:0009. Thus, the assembly-language source for an expansion ROM would look like this:

```
expansion   segment
e_seq       db          "EXPANSION" ; must be upper-case.

e_init      proc        far         ; initialization routine
            .....                   ; Do initialization
            ret         ; Pass control back to monitor
e_init      endp

;;; Place rest of expansion ROM code here ;;;
expansion   ends
            end
```

Note that in this example, control is returned to the monitor once the expansion code has initialized itself. It may be desirable not to pass control back to the monitor. In this case, the expansion ROM "takes over" the system. This could be useful in applications where the user should not be able to see the code or modify memory, but only perform specific actions defined by the expansion ROM.

5.2. *Initialization*

The sequence of startup is as follows:

- A RAM test is performed on the bottom 2K of RAM.
- The monitor initializes its operating parameters.
- The 8259 Interrupt Controller is initialized.
- The 8254 Timer is initialized.
- The 8251A USART is initialized.
- A lamp test is performed.

- If an expansion ROM is found, control is passed to it. The segment registers DS and ES point to the monitor's data area upon entry to the expansion ROM initialization routine. These registers should be preserved by the initialization routine. The initialization routine should be declared as a FAR procedure.

- Otherwise (or after the ROM initialization routine returns), the main command entry loop is entered.

5.3. Threading in the Expansion ROM

To thread the expansion ROM into the system, certain interrupt vectors must be initialized. If the expansion ROM wishes to add commands to the monitor, it must contain two command tables. The first is the name table. Each entry in the table has the following format:

byte 0 - a number indicating the smallest number of characters which must match for the command to be selected.

bytes 1 to n - a zero-terminated, upper-case string representing the command name.

The end of the table is signified by a zero byte in the "number of characters to match" field.

For example, suppose we wanted to add two new commands, FOOBAR and WIDGET. Suppose we wanted at least F to be typed for FOOBAR, and WID for WIDGET. The name table would look like this:

```
namtab          db    1     ; Match at least 1 char
                            ; for FOOBAR
                db    "FOOBAR", 0

                db    3
                db    "WIDGET", 0

                db    0     ; end of table.
```

The second table that is required is a table of offsets for the command routines to be called when a command is activated. This is called the command vector table. Note that this table AND the command execution routines MUST BE IN THE SAME SEGMENT AS THE NAME TABLE. In this example, the command vector table might look like this:

```
vectab          dw    foobar_cmd
                dw    widget_cmd
```

FOOBAR_CMD and WIDGET_CMD are routines which are called when the Foobar or WIDget commands are entered.

To thread in these tables, the expansion ROM initialization routine should set interrupt vector 17 decimal to point to the name table. This is in the standard 8088 offset-lo, offset-hi, segment-lo, segment-hi order. Note that the offset MUST NOT be 0000, or else the monitor will not recognize the table. This is not a problem, since the table is relative to the start of the expansion ROM, and offset 0 of the ROM is already in use by the signature sequence. The first two bytes of interrupt vector 18 decimal should be set to the offset of the command vector table. It is assumed that this table is in the same segment as the name table; thus, the segment field of interrupt vector 18 is NOT USED and is IGNORED. For the example shown above, the following code will accomplish the initialization:

```
ints                segment at 0
                    org   44H               ; interrupt 17
e_cmd_names         dd    ?
e_cmd_vec           dd    ?

ints                ends

expansion   segment
;;;         Signature bytes, etc. ;;;
            mov         word ptr e_cmd_names, offset namtab
            mov         word ptr e_cmd_names+2, seg namtab
            mov         word ptr e_cmd_vec, offset vectab
;;;         Rest of code      ;;;
expansion   ends
```

5.4. Command Execution Routines

The command execution routines are the routines which are called when a command is being executed. They must follow certain rules:

- They must be FAR procedures.

- They should return control to the monitor with a far return or a call to FN_ERROR.

Upon entry to a command execution routine, the state of the machine is as follows:

- DS, ES point to the monitor's data segment

- SS:SP point to the current top of the monitor's stack

- ZF is set if no parameters followed the command name; otherwise, ZF is cleared.

- If there are parameters, then DS:SI point to the first character of the first parameter. The list of parameters is a zero-terminated string which has been processed by FN_NICELIN. Parameters are separated by single spaces.

For example, the FOOBAR_CMD routine might look like this:

```
fn_led      equ   6           ; Function call


foobar_cmd  proc  far
            je    no_params   ; ZF set if no params
```

```
                mov   ch, fn_led
                mov   al, 0
                int   7           ; turn off all LEDs
                ret

    no_params:  mov   ch, fn_led
                mov   al, 0FFH
                int   7           ; Light all LEDs
                ret
    foobar_cmd  endp
```

This simple example lights the LEDs if there are no parameters, and extinguishes them if there is at least one parameter.

5.5. Software Hooks

The MUN-88 Monitor contains an additional mechanism for enhancing the existing code. This mechanism, known as software hooks, allows the existing HELP command to be upgraded, as well as allowing the single-step or breakpoint display to be enhanced.

Interrupt vector 16 decimal is defined as the hook vector. It is initialized by the monitor to point to a null routine. The expansion ROM may initialize this vector to point to its own hook handling routine, and thus modify the HELP command and/or single-step display. The hook-handling routine should preserve the segment registers, but may modify other registers. It should return control to the monitor with an IRET instruction.

There are two software hooks currently defined. Each hook is given a number. Prior to executing an INT 16, the monitor loads the appropriate number into the CH register, and sets up any other registers as described for each hook. The hooks are as follows:

Number: 0 HELP_HOOK

This hook is executed after the last line of display from the HELP command has been sent to the terminal. This allows the expansion ROM to then print its own help information.

Number: 1 DISP_HOOK

This hook is executed immediately after the single-step or breakpoint display. Register AX contains the segment of the instruction about to be executed, and DX contains the offset. This hook was added in anticipation of the writing of an assembler/disassembler. Thus, the assembly-language mnemonic can be printed along with the other information in the single-step/breakpoint display.

5.6. Example of a hook-handler

This simple example adds a help message when HELP is executed. It ignores the DISP_HOOK hook.

```
; Hook vectors
hooks segment     at 0
            org         40H         ; INT 16 is hook vector
hookvec     dd          ?
hooks ends

; Initialization code
code        segment
            assume      cs:code, ds:hooks

; Point DS to hook vector segment
            mov         ax, hooks
            mov         ds, ax

; Initialize the vector
            mov         word ptr hookvec, offset h_handler

; The hook handler must be in the same code segment as
; this code! We use "cs" instead of "seg h_handler" so
; that binary image is relocatable.
            mov         word ptr hookvec+2, cs

; End of initialization of hook handler
; ; ; MORE CODE, ETC    ; ; ;

; This is the actual hook handler routine.

h_handler   proc        far
            cmp         ch, 0 ; Is it HELP_HOOK?
            je          continue    ; yes, do it
            iret                    ; no, ignore it - exit
continue:
            push        ds          ; save DS
            push        cs
            pop         ds          ; point DS to code segment
            mov         di, offset message
            mov         ch, 4 ; FN_PUTLIN
            int         7           ; print the message
            pop         ds          ; restore DS
            iret                    ; return to monitor
h_handler   endp
message     db          "This is the new help line."
            db          0DH, 0AH    ; CR, LF
            db          0           ; Terminate the string
; ; ; MORE CODE, ETC ; ;

code        ends
```

6. The Monitor Simulator

An additional version of the MUN-88 monitor is available which runs on the IBM PC or compatibles. The program is called MONITOR.EXE, and is started by typing MONITOR from the DOS prompt. This program simulates the environment of the MUN-88 as much as possible. All function calls are available, and hooks may still be used. However, there are some differences which must be taken into account. To exit from the monitor simulator, press ESC in response to the dot prompt.

When the simulator starts, the register images are set up as follows:

- CS:IP point to the first instruction in the monitor simulator. The actual values depend on where the simulator was loaded.

- SS:SP point to the top of the user stack space. The monitor reserves 64 words (128 bytes) of stack space for the user. If you need more, you must set up your own stack.

- ES:0000 points to the first free memory location. ES does NOT necessarily contain the value 0032; the actual value depends on where the monitor simulator was loaded.

- DS points to the monitor's data segment. The actual value depends on where the monitor was loaded.

- All other registers contain 0000, and all flags are cleared.

Note that 8K of memory is free for user-written programs, starting from ES:0000. Thus, all of the code for an expansion ROM can be loaded into the simulator.

The DLOAD command is modified slightly - in addition to a seg:off parameter, it requires a filename parameter. The filename and extension must be fully specified. This command loads the specified file into memory. For example:

```
            DLOAD ES:0 TEST.BIN

            dl 285C:0 c:\mydir\test2.tst
```

Obviously, the LED display is unavailable on the IBM PC. The raising of alarms, and the LED command simply modify the RAM image of the LEDs; no other action takes place. This is another reason why user-written programs should use the LED function calls rather than directly accessing the LED port - the first method will work on the simulator, whereas the second could cause the PC to crash.

The hook interrupt vectors are moved on the simulator so as not to conflict with BIOS interrupts. The vectors INT 16, INT 17 and INT 18 are redefined as INT 96, INT 97 and INT 98 on the simulator.

6.1. The MUN88.H File

The following file is called MUN88.H. If you INCLUDE it in your assembly-language programs, it will make moving between the monitor and the simulator much easier. Here is how you would use the file:

```
      TRUE        equ   -1
      FALSE equ   0

      ; The following symbols MUST be defined before including
      ; MUN88.H. If you are assembling a program to run on
      ; the simulator, use the values shown below. If you are
      ; assembling a program to run on the actual MUN-88,
      ; exchange the TRUE and FALSE values.
      ibmpc equ   TRUE
      mun88 equ   FALSE

                  include     MUN88.H
```

If you use the MUN88.H file, and refer to hook vectors, function calls, etc. by the symbols found in the file instead of by absolute address, then you only need to set the ibmpc and mun88 flags as appropriate to assemble your program for the simulator or the actual MUN-88 Monitor. All the other steps in creating a binary file remain the same.

6.2. Listing of the MUN88.H File

```
;************************************************************
;*                                                          *
;* M U N - 8 8  H E A D E R  F I L E - USEFUL CONSTANTS     *
;*                                                          *
;* FOR MUN-88 SINGLE-BOARD COMPUTER                         *
;*                                                          *
;* BY DAVID SKOLL AND JOSEPH LIANG                          *
;*                                                          *
;* COPYRIGHT (C) D. SKOLL AND J. LIANG 1988                 *
;*                                                          *
;************************************************************

; Control characters
cr    equ 0DH    ; Carriage Return
lf    equ 0AH    ; Line Feed
bel   equ 07H    ; Bell character
_esc  equ 1BH    ; ESC
stx   equ 02H    ; STX
etx   equ 03H    ; ETX
bs    equ 08H    ; BS
```

```
; Timing constants
tenms equ 2805   ; Constant for delay of 10 ms

; Alarms

sw_err_alm         equ 0  ; software error
xmit_timeout_alm  equ 1  ; transmit timeout
recv_err_alm      equ 2  ; receive error
input_over_alm    equ 3  ; input buffer overflow
line_long_alm     equ 4  ; line truncated
bad_fn_call_alm   equ 5  ; bad value in CH during function call interrupt
bad_hook_alm    equ 6 ; bad value in CH during hook (raised by expansion ROM)
unknown_int_alm   equ 7  ; unknown interrupt received

; Function calls available to user

fn_getc           equ 0  ; get a char from terminal
fn_getc_nowait    equ 1  ; get a char from terminal if one is ready
fn_getlin         equ 2  ; get a line of input from terminal
fn_putc           equ 3  ; send a char to terminal
fn_putlin         equ 4  ; send a line to terminal
fn_nicelin        equ 5  ; prepare a line for easy parsing
fn_led            equ 6  ; set LED display
fn_led_on         equ 7  ; turn on specified LEDs
fn_led_off        equ 8  ; turn off specified LEDs
fn_qalm           equ 9  ; query alarms
fn_clralm         equ 10  ; clear alarms
fn_raise_alarm    equ 11  ; raise an alarm
fn_flush          equ 12  ; flush the terminal input buffer
fn_getnum         equ 13  ; parse a number
fn_putnum_byte    equ 14  ; convert a byte number to ASCII
fn_putnum_word    equ 15  ; convert a word number to ASCII
fn_putseg_off     equ 16  ; convert a seg:off to ASCII
fn_alm            equ 17  ; set or query LED annunciator status
fn_error          equ 18  ; take the error exit path
fn_getval         equ 19  ; parse a value (can be a register name)
```

```
fn_getseg_off     equ 20  ; parse a seg:offset parameter
fn_getval_chk     equ 21  ; parse a value; take error exit path if error
fn_crlf           equ 22  ; send a CR/LF sequence to terminal
fn_cond_crlf      equ 23  ; send a CR/LF if cursor is not in left-most column
fn_tab            equ 24  ; tab cursor to specified column
fn_lower_alarm    equ 25  ; clear a specified alarm

; The hook values which are currently defined
help_hook     equ 0   ; Executed after HELP messages printed
disp_hook     equ 1   ; Executed after single-step or breakpoint
                     ; display

;I/O ports
;  LED and SWITCH port:

  ledport equ 30h
  swport  equ 30h

;  8259A Interrupt Controller

  int_ctrl0 equ 00
  int_ctrl1 equ 01

;  8251A USART

  usart_data  equ 10H
  usart_status equ 11H
  usart_ctrl  equ 11H

;  8254 Programmable Interval Timer

  timer0   equ 20H
  timer1   equ 21H
  timer2   equ 22H
  timer_ctrl equ 23H

;
```

```
; Interrupt vectors
;

intvec segment at 0

int0_vec  dd   ? ; INT 0 Divide by Zero (Not implemented)
int1_vec  dd   ? ; INT 1 Single Step
int2_vec  dd   ? ; INT 2 NMI (Not used)
int3_vec  dd   ? ; INT 3 Breakpoint
int4_vec  dd   ? ; INT 4 Overflow (Not used)
int5_vec  dd   ? ; INT 5 Not used
int6_vec  dd   ? ; INT 6 Program Terminate
int7_vec  dd   ? ; INT 7 Monitor function call
int8_vec  dd   ? ; INT 8 Real-time clock (Not implemented)
int9_vec  dd   ? ; INT 9 Reserved
inta_vec  dd   ? ; INT 10 USART receiver interrupt
intb_vec  dd   ? ; INT 11 Reserved
intc_vec  dd   ? ; INT 12 May be used by additional devices
intd_vec  dd   ? ; INT 13 May be used by additional devices
inte_vec  dd   ? ; INT 14 May be used by additional devices
intf_vec  dd   ? ; INT 15 Reserved

; Now, the hooks for expansion ROM
   IF ibmpc
        org 180H         ; So that hooks don't conflict with DOS
hook_int  equ 96
   ENDIF

   IF mun88
hook_int  equ 16
   ENDIF

hookvec     dd   ? ; INT 16 the hook interrupt vector      INT 96 on IBM
e_cmd_names dd   ? ; INT 17 expansion ROM command names     INT 97 on IBM
e_cmd_vec   dd   ? ; INT 18 expansion ROM command vector table INT 98 on IBM
e_int19     dd   ? ; INT 19 reserved for future use by monitor INT 99 on IBM
e_int20     dd   ? ; INT 20 reserved for future use by monitor INT 100 on IBM
```

```
e_int21     dd   ? ; INT 21 reserved for use by expansion ROM  INT 101 on IBM
e_int22     dd   ? ; INT 22 reserved for use by expansion ROM  INT 102 on IBM

intvec ends
```

APPENDIX A: Using KERMIT with the MUN-88 Monitor

The public-domain program KERMIT may be used to communicate with the MUN-88 Monitor. To make it easier to use, set up the following file, called MUN88.KRM:

```
set key \270 \008

set speed 4800

set parity none

set terminal none

connect
```

Then, whenever you want to use the MUN-88 Computer, set the speed on the computer to 4800 baud, start the KERMIT program, and type:

```
      TAKE MUN88.KRM
```

to begin your session.

To download a file to the MUN-88 Computer, type the normal DLOAD command and press ENTER. Then, escape from your connection in KERMIT (^] C) and type:

```
      RUN MUN88DL filename
```

where filename is a binary image which you wish to download to the MUN-88 Computer. You must fully specify the filename, including the extension.

Once the KERMIT prompt returns, type CONNECT to resume communication with the MUN-88 Computer, and press ENTER to see the results of the download command. The file MUN88DL.EXE is created from MUN88DL.C, a program written in Turbo C. The listing of MUN88DL.C is given in Appendix B.

APPENDIX B: MUN88DL.C

```
 /**********************************************************
 *                              *
 *  Program used to download a file to the MUN-88     *
 *  single-board microcomputer              *
 *                              *
 *  Copyright (C) 1988 by David F. Skoll         *
 *                              *
 **********************************************************/

#include <stdio.h>
#include <bios.h>
#include <string.h>
#define PORT 0
#define STX 2
#define ETX 3
#define ESC 27

void send_serial(char c);

void main(int argc, char *argv[])
{
  int i;
  unsigned int checksum;
  int c;
  FILE *fp;

  if (argc != 2)
  {
   fprintf(stderr, "Usage: %s filename\n", argv[0]);
   exit(1);
  }

  if ((fp = fopen(argv[1], "rb")) == NULL)
  {
   fprintf(stderr, "Could not open file %s.\n", argv[1]);
   exit(1);
  }

  checksum = 0;

  /* Send the STX char to indicate file to be sent. */

  send_serial(STX);
  while (! feof(fp))
  {
   c = fgetc(fp);
   if (c != EOF)
   {
     if (c == ETX || c == ESC) send_serial(ESC);
     send_serial(c);
     checksum += (unsigned int) c;
   }
  }

  fclose(fp);

  /* Send the ETX */
  send_serial(ETX);

  /* Send the checksum */
  send_serial(checksum & 0xFF);    /* Lo-order byte */
  send_serial((checksum >> 8) & 0xFF); /* Hi-order byte */
  return;
}


void send_serial(char ch)
{
  while((bioscom(3, 0, PORT) & 0x2000) != 0x2000)
   ;
  bioscom(1, ch, PORT);
}
```