INTRODUCTION TO THE UNIX OPERATING SYSTEM

This manual contains the entire set of documents describing the UNIX operating system. This particular version is the fourth in a series of UNIX look-alike systems built during the last five years. It is intended for a single user and does not support any multi-tasking or multi-user features. It is simple to mount and use on Z-80 systems having 64 Kbytes of RAM and an 8086 version will be available shortly. The operating system can access different types of storage without modification.

It supports a hierarchial file system complete with pathnames and subdirectories. Files are placed in the filesystem using dynamic allocation, can be protected and can reach a maximal length of 8 Mbytes. Files can be declared to be physical device drivers allowing uniform treatment of both disk files and real devices such as printers and modems. The operating system supplies several subroutines for reading and writing bytes or blocks of bytes to or from files. In doing so, no constraints are made on the data allowing arbitrary binary patterns on all 8 bits. It allows simultaneous read and write access of the same file and permits up to eight files to be open simultaneously. Entry points for moving the file pointers are provided for allowing random access of any byte within any file. Volumes of the filesystem can be mounted and removed at will allowing the user to access volumes as subdirectories.

There is a shell programming language which allows the passing of arguments to commands as well as re-direction of the standard I/O channel. Pseudo pipes are provided to pass I/O between commands without user intervention. The shell allows multiple commands on a single line and is easily made to read files containing command lines. Such files can be declared executable which will cause the operating system to interpret their content as commands whenever the file name is entered as a command. This allows the user to quickly make new commands which themselves are built from other commands.

The operating system also includes entry points for printing and reading numbers or strings. It allows tab stops to be set and provides a string reading routine which recognizes character and line delete. It also includes a string comparator which matches two text lines.

There are numerous commands mounted under the operating system for
creating, manipulating and removing files or directories. Also mounted are code generation systems for both the Z-80 and 8086 processors. These each include a C compiler, an assembler producing relocatable code and a link editor for combining modules which were generated separately. A screen oriented text editor as well as a text processor are included for building programs and doing word processing.

Thus, the UNIX operating system provides all necessary tools for generating quality software. It has been subjected to intense use on several machines for nearly two years. Several applications have been developed using it including a bit-slice machine programmed entirely in microcode designed on this system. The 8086 software generation system mentioned earlier was developed in its entirety using this operating system on a Z-80.
MOUNTING THE UNIX OPERATING SYSTEM

This note describes a procedure for mounting the UNIX operating system on your computer hardware. It is expected that the system consists of a Z-80 processor and 64 Kbytes of RAM. While it is possible to run the system with less RAM, it is not recommended. To determine whether your available RAM is adequate, a memory map is attached which shows the purposes for which various blocks of memory is used.

It is necessary for you to develop disk driver routines which are compatible with an 8 inch disk using single density sectors having 256 bytes per sector and 15 sectors per track. This is usually a simple modification of routines which already exist within your system. Generally, the only modification required is the number of bytes read from the disk controller chip, usually a 1771 or 1791.

Once this is done, you should load the UNIX module from the system disk. The module is located on 64 consecutive sectors starting with track 0, sector 4. This amounts to exactly 16 Kbytes. To check your loading procedure, a hexadecimal dump of the first of these sectors is attached.

Next, you should construct the "bios" (Basic I/O Software) for your hardware. To assist you in this task, a sample bios is also attached. Notice that only five routines are required. Two of these are the console input and output routines. It is recommended that the general structure of the conin and conout routines in the sample bios be preserved. Simple modifications of these will enable you to use different ports or to call subroutines for the console I/O such as would be required for a video RAM. Notice that there are sections which check for certain control characters (cntl-q, cntl-s, cntl-x) for restarting the operating system. It is important that these sections remain as they are.

The remaining routines are for interfacing the disks in your system. There are three such routines which read sectors, write sectors and format the disk. Two parameters are passed to the disk drivers by the operating system. The first of these is the memory address where the
256 byte memory block is located which is to be read or written. The second is an integer between 0 and 32767 which is referred to as an inode. The address is passed to the routines in the HL register. The inode is the second value on the stack, the first being the return address which must be preserved. This inode integer must be decomposed into the disk, track and sector which you wish it to reference. Normally, the value is decomposed as shown below:

\[
\text{disk id} = \text{high}_\text{byte}(\text{inode}/15);
\text{track id} = \text{low}_\text{byte}(\text{inode}/15);
\text{sector id} = 1+\text{remainder}(\text{inode}/15);
\]

Before you panic and begin writing a divide and remainder program, let me inform you that the operating system includes one! The sample bios shows a simple program which unravels the stack to get the address and inode and goes on to convert it into disk/track/sector. If the disk system you are using supports something other than 15 sectors per track, it is only necessary to modify the parameter "15" before calling the divide/remainder routine.

The third disk driver is a formatter which should re-write the disk drive named in the L register on entry. To obtain the greatest performance speed from the operating system, the sequence of sector numbers given in the sample bios should normally be used. If an intelligent disk controller is used, the sequence given may need to be modified. The scheme here is to interleave the sectors on the disk so that the "read" subroutine (see the subroutine manual) wastes the least amount of time possible between sector transfers. This is determined by experimenting a large file into memory and timing the load interval. Then, change the interleave sequence and repeat the experiment. Continue until the load time appears optimized. In any event, the formatter is not needed when the system is first brought up and in no event is the sequence given a requirement to make the system operational. Any interleave sequence can be used initially and optimization should wait until the operating system is functional.

One final note on the disk driver routines. Each routine should detect errors in accessing the disk drives and should make several attempts to accomplish the transfer. If the routine fails to make the required transfer, do not simply return to the operating system. Instead, print the command, disk drive, track, sector and memory address and jump to the warm start entry point (OC003H). This will allow the operating system to recover gracefully. If desired, the programs could simply "hang" in an infinite loop after an error. Then, the operator could remove the disk and reset the system manually. Either is acceptable.

After the five routines are built for location OC057H and the 64 sectors of the operating system have been successfully loaded, you should simply merge the two by first loading the operating system at
location OC000H and then overlaying the new bios at location OC057H. Note that the new bios cannot extend beyond location OC51AH. Next, simply place the operating system disk in drive 0 and jump to location OC000H. After a few disk accesses, the system will type its cold start message and you are ready to go! A later section of this manual will describe a few commands useful for getting started. After you are sure the bios is working properly, you should re-write the 64 sectors starting at track 0, sector 4 with the modified copy of the operating system. After this is done, you can simply load the block at memory location OC000H and jump to the beginning of the system.
THE FOLLOWING EQUATES ARE FOR AN 8251 AT I/O PORT 0. THE MASK WORDS ARE USED TO DETERMINE THE STATE OF THE UART USING THE STATUS PORT.

DATA EQU 0 ; I/O PORT FOR UART DATA
STATUS EQU 1 ; I/O PORT FOR UART STATUS
IMASK EQU 02H ; DATA AVAILABLE MASK FOR INPUT
OMASK EQU 01H ; BUFFER EMPTY MASK FOR OUTPUT

THE JUMP TABLE WHICH FOLLOWS MUST BE PLACED AT LOCATION 0C057H IN THE OPERATING SYSTEM. THE ENTIRE BIOS PACKAGE MUST FIT IN THE BLOCK OF MEMORY FROM 0C057H TO 0C51AH. THESE FIVE ROUTINES ARE ALL THAT IS REQUIRED FOR THE USER TO INTERFACE THE OPERATING SYSTEM.

ORG 0C057H

JMP CONIN
JMP CONOUT
JMP LOAD
JMP SAVE
JMP FORMAT

CONSOLE INPUTING ROUTINE

USED TO READ CHARACTERS FROM THE CONSOLE DEVICE AND RETURN THEM TO THE OPERATING SYSTEM. IT CHECKS FOR SEVERAL SPECIAL CHARACTERS WHICH CAN BE TYPED BY THE USER FOR RESTARTING THE OPERATING SYSTEM. THESE ARE:

CNTL-X (18H) COLD START (0C000H)
CNTL-Q (11H) WARM START (0C003H)
NULL (00H) END OF FILE, TRANSLATE TO 8000H
CNTL-S (13H) IGNORE
LINEFEED (0AH) TRANSLATE TO CARRIAGE RETURN (0DH)

CONIN: IN STATUS ; CHECK FOR DATA AVAILABLE
ANI IMASK ; USING THE MASK
JZ CONIN ; WAIT UNTIL A CHARACTER ARRIVES
IN DATA ; GET IT
ANI 7FH ; MASK PARITY
MOV L,A ; PUT IT IN THE L REGISTER
MVI H,0 ; CLEAR THE H REGISTER
CPI 0AH ; TEST FOR LINEFEED
JZ NLINE ; IF SO, TRANSLATE TO CAR. RET.
CPI 00H ; TEST FOR END OF FILE
JZ EOFILE ; IF SO, TRANSLATE TO 8000H
CPI 13H ; TEST FOR CNTL-S
JZ CONIN ; IF SO, IGNORE
CPI 11H ; TEST FOR CNTL-O
JZ 0C003H ; IF SO, WARM START
CPI 18H ; TEST FOR CNTL-X
JZ 0C000H ; IF SO, COLD START
RET ; NOTHING SPECIAL, QUIT

INE: MVI L,ODH ; FORCE VALUE TO CAR. RET.
RET ; QUIT

EOFILE: MVI H,80H ; FORCE VALUE TO 8000H (EOF)
RET ; QUIT

CONSOLE OUTPUTING ROUTINE

; THIS ROUTINE TRANSMITS A CHARACTER TO THE CONSOLE
; WHEN CALLED BY THE OPERATING SYSTEM. IT IS REQUIRED
; TO TRANSLATE CARRIAGE RETURN (ODH) INTO BOTH CARRIAGE
; RETURN AND LINEFEED (OAH). IF A CARRIAGE RETURN IS
; TO BE SENT, IT CHECKS TO SEE IF ANY SPECIAL CHARACTERS
; HAVE BEEN SENT BY CALLING THE CHKIO PROGRAM.

CONOUT: MOV A,L ; CHAR TO TYPE IN L ON ENTRY
CPI ODH ; TEST FOR CAR. RET.
JNZ CONX ; IF NOT, JUST SEND THE CHAR
CALL CHKIO ; TEST FOR RESTARTS OF THE OP. SYS.
MVI L,OAH ; SET UP FOR LINE FEED
CALL CONX ; SEND IT
MVI L,ODH ; SET UP FOR CAR. RET. AND SEND IT

CONX: IN STATUS ; TEST THE UART FOR
ANI OMASK ; TRANSMIT BUFFER EMPTY
JZ CONX ; WAIT UNTIL NOT BUSY
MOV A,L ; GET THE CHAR TO SEND FROM L
OUT DATA ; SEND IT
RET ; QUIT

CHKIO TESTS THE RESTART STATUS OF THE OP. SYS.

; THERE IS A VARIABLE AT LOCATION 9FD6H WHICH
; DETERMINES IF RESTARTS ARE ALLOWED OR NOT.
; IF THE VARIABLE IS 0, THEY SHOULD BE ALLOWED.
; THIS ROUTINE CHECKS TO SEE IF THE VARIABLE
; IS IN A STATE ALLOWING COLD OR WARM STARTS TO
; BE MADE. IF ENABLED, IT THEN TESTS TO SEE IF
; THE CONSOLE HAS TYPED A CNTL-X (COLD START),
; OR A CNTL-Q (WARM START). IT ALSO CHECKS FOR
; CNTL-S WHICH IS USED TO TEMPORARILY STOP OUTPUT
; AT THE CONSOLE.

CHKIO: LDA 9FD6H ; THIS VARIABLE SHOULD BE TESTED
ORA A ; FOR ANY NON-ZERO VALUE
RNZ ; IF NON-ZERO, QUIT
IN STATUS ; OTHERWISE, CHECK FOR DATA AVAILABLE
ANI IMASK ; IN THE UART INPUT SIDE
RZ ; IF NONE, QUIT
IN DATA ; ELSE, READ THE DATA
ANI 7FH ; MASK PARITY
CPI 11H ; TEST FOR CNTL-Q
JZ 0C003H ; IF SO, WARM START
CPI 18H ; TEST FOR CNTL-X
JZ 0C000H ; IF SO, COLD START
CPI 13H ; TEST FOR CNTL-S
JZ HOLD ; IF SO, HOLD ALL OUTPUT
RET ; NOTHING, SO QUIT

LOAD: IN STATUS ; TEST FOR A CHARACTER
ANI IMASK ; IN THE UART
JZ HOLD ; WAIT UNTIL IT ARRIVES
RET ; QUIT WITHOUT READING IT

LOAD ROUTINE TRANSFERS FROM DISK TO MEMORY

THIS ROUTINE IS PASSED AN INODE VALUE WHICH
IT DECOMPOSES INTO DISK/TRACK/SECTOR. IT IS
ALSO PASSED THE ADDRESS WHERE IT IS TO PLACE
THE SPECIFIED SECTOR. IF DIFFERENT DRIVES
ARE TO BE MOUNTED, THIS ROUTINE MUST BE
MODIFIED TO INTERFACE THEM.

LOAD: XCHG ; LOAD ADDRESS INTO DE
POP H ; GET THE RETURN ADDR
XTXL ; RESTORE IT, GET THE INODE
PUSH D ; SAVE ADDRESS ON STACK
LXI D,15 ; SET DE TO SCTRS PER TRACK (15)
CALL 0C012H ; DIVIDE, REMAINDER FUNCTION
INR E ; NOW, E IS THE SECTOR TO LOAD
MOV D,L ; NOW, D IS THE TRACK TO LOAD
MOV C,H ; NOW, C IS THE DISK TO ACCESS
POP H ; NOW, HL IS THE ADDRESS TO LOAD

PLACE YOUR DISK DEPENDENT PROGRAMS HERE. IT
SHOULD TRANSFER THE DISK/TRACK/SECTOR SPECIFIED
BY THE ABOVE REGISTERS TO THE MEMORY POINTED TO
BY THE HL REGISTER. IT SHOULD MAKE SEVERAL TRIES
TO LOAD THE SECTOR WHICH IS 256 BYTES IN LENGTH.
IF IT IS NOT SUCCESSFUL, IT SHOULD REPORT THE ERROR
AND THEN JUMP TO THE WARM START ENTRY POINT (0C003H).
IF NO ERRORS WERE MADE, IT SHOULD JUMP TO THE
CHKIO ROUTINE TO ALLOW SYSTEM RESTARTS TO BE MADE.

JMP CHKIO ; CHECK FOR RESTARTS

SAVE ROUTINE TRANSFERS FROM MEMORY TO DISK

THIS ROUTINE IS PASSED AN INODE VALUE WHICH
IT DECOMPOSES INTO DISK/TRACK/SECTOR. IT IS
ALSO PASSED THE ADDRESS WHERE IT IS TO FIND
THE SPECIFIED SECTOR. IF DIFFERENT DRIVES
ARE TO BE MOUNTED, THIS ROUTINE MUST BE
MODIFIED TO INTERFACE THEM.

SAVE: XCHG ; LOAD ADDRESS INTO DE
POP H ; GET THE RETURN ADDR
XTXL ; RESTORE IT, GET THE INODE
PUSH D ; SAVE ADDRESS ON STACK
LXI D,15 ; SET DE TO SCTRS PER TRACK (15)
CALL 0C012H ; DIVIDE, REMAINDER FUNCTION
INR E ; NOW, E IS THE SECTOR TO LOAD
MOV D,L ; NOW, D IS THE TRACK TO LOAD
MOV C,H ; NOW, C IS THE DISK ID AT HL.
POP H ; NOW, HL IS THE ADDRESS TO LOAD

PLACE YOUR DISK DEPENDENT PROGRAMS HERE. IT SHOULD WRITE THE DISK/TRACK/SECTOR SPECIFIED BY THE ABOVE REGISTERS WITH THE MEMORY POINTED TO BY THE HL REGISTER. IT SHOULD MAKE SEVERAL TRIES TO WRITE THE SECTOR WHICH IS 256 BYTES IN LENGTH. IF IT IS NOT SUCCESSFUL, IT SHOULD REPORT THE ERROR AND THEN JUMP TO THE WARM START ENTRY POINT (0C003H). IF NO ERRORS WERE MADE, IT SHOULD JUMP TO THE CHKIO ROUTINE TO ALLOW SYSTEM RESTARTS TO BE MADE.

JMP CHKIO ; CHECK FOR RESTARTS

FORMAT ROUTINE FORMATS AN ENTIRE DISK FOR 256 BYTE SECTORS, SINGLE DENSITY

FORMAT: MOV A,L ; GET THE DISK ID INTO A

PLACE YOUR DISK FORMATTER PROGRAM HERE. THE DISK TO FORMAT IS IN THE L AND A REGISTERS.
ALL TRACKS SHOULD BE FORMATTED. WHILE ANY INTERLEAVE OF SECTORS IS ALLOWED, THE SYSTEM WILL PERFORM FASTEST IF THE FOLLOWING INTERLEAVE IS USED.

EVEN TRACKS: 1 9 2 10 3 11 4 12 5 13 6 14 7 15 8
ODD TRACKS: 5 13 6 14 7 15 8 1 9 2 10 3 11 4 12

THIS SELECTION MAY REQUIRE MODIFICATION IF AN INTELLIGENT DISK CONTROLLER SUBSYSTEM IS USED.
A GUIDE TO THE UNIX OPERATING SYSTEM

These notes are intended to help the user understand the operation of UNIX. It attempts to explain how the system operates and what capabilities it has. Detailed information for each command is contained in the Command Set section.

THE FILE SYSTEM

The UNIX file system is, perhaps, its most attractive feature. It supports a hierarchal file system under which directories can have subdirectories which can have subdirectories ad infinitum. When the system is booted, it begins in the "root" directory. This is the highest level in the directory tree. Within the root directory, there can be any number of subdirectories. To find out if there are any subdirectories in the root, type "ls -al". In the output, the leftmost columns contain the mode of each file. Those having a "d" at the end of their mode are directories. You will see that "bin" and "lib" are directories. To move inside a directory, type "cd bin". Now, type "ls -al" and you will see a different set of files entirely. To get back to the root, type "cd". As you can see, the cd command allows you to move about in the directory tree. Whatever directory you are in at any given time is referred to as the "current" directory. It can be referenced in names as ".". The directory which contains the current directory as a subdirectory is referred to as the "parent" directory. It can be referenced in names as "..". A "grandparent" directory can be referenced using "../..". Any command which references a file will look for the filename in the current directory unless you specify otherwise. Using the characters "/" and "..", it is possible to reference files in other parts of the filesystem. For example, the file names "hello", "/hello" and "../hello" can reference three different files of the same name (hello). The first of these would be found in the current directory. The second would be in the root directory and the third would be found in the parent. It is possible to reference files in subdirectories below the current directory by names such as "dirx/hello". Other possibilities such as "../dirx/temp/hello" exist. This notion of referencing files in other than the current directory is referred to as "pathname". A pathname is the path to a file such as "/dir1/dir2/file" in which case the path begins in the root (due to the leading slash) or "../dirx/file" in which case the path begins in the current directory (due to no leading slash). These two referencing techniques are
referred to as "absolute" path (/bin/temp) and "relative" path (./dirx/hello).

It is possible to make new directories (see mkdir), remove them (see rmdir) and move them about (see mvdir). There is no limit on the number of subdirectories that can exist within any single directory or the total number of them within the system. Directory names (and ordinary files as well) can be made to disappear in listings of the directory in which they appear (see chmod).

Ordinary files can assume any length up to 8 Mbytes which is the maximum addressing range of the filesystem. Filenames can consist of any alphabetical character, upper or lower case, any number and the characters "." and "_". Names are restricted to 13 characters. Files can be protected from reading or writing and they can be made to not list when the directory content is listed. Files can be moved about (see mv), copied (see cp), removed (see rm) or transferred to other disks (see uucp, dup). Unlike other filesystems, there is no filetype for determining the use of the file and for that reason, there is no restriction as to the way in which files are used. Naming conventions are generally used to differentiate between files that are C programs (name.c), assembly language programs (name.s), link modules (name.o) and executable routines (name). However, these are simply conventions and are not required for the proper operation of the filesystem or the operating system. There is no constraint on the content of the files and any 8 bit pattern can be passed to or from them. End-of-file (EOF) is denoted by the value 0x0000 (0:0000 in C) which cannot be confused with any 8 bit data pattern. Operating system entry points are provided for reading (getc, getchar, read) and writing (putc, putchar, write) files. Another entry point, seek, allows the user to move the file pointers allowing random access of any byte within a file. Like directories, there is no constraint on the number of files in the filesystem.

RUNNING PROGRAMS

After uNIX is booted, it checks to see if there is a file named "profile" in the root directory. If there is, it will execute it as a set of system commands. This is the source of the login message. If you wish to set parameters or run programs you have developed each time the system is booted, you can simply place the appropriate command line in the "profile" and the system will execute it each time it is booted. The profile is list and write protected. To list it, use "ls -a" and to see its content, use "cat profile" or "vi profile". To modify it, you must force the file mode bits to 0 using "chmod".

To run a program, simply type its name. The operating system will search a set of directories specified by the "path" variable for the name. If found, it will load the program and execute it. The path variable can be displayed and modified using the stty command. Further details on it can be found in the system subroutine. The default value of the path variable will cause the system to first search in the "/bin" directory and then in the current directory. The bin directory holds all operating system commands. When entering commands, multiple
commands can be entered on a single line from the console if they are separated by semicolons (;).

STANDARD I/O

When any program runs, any output it sends using putchar and any input it reads using getchar can be directed to or from a file (instead of the terminal) when the command is entered. For instance, the command "ls" sends its output to the standard output (virtually every command does). If it is desired to save the listing in a file, simply type "ls > filename" instead. If you wish to add more output to the end of a file, an append operator is provided. For example, "ls -l >> filename" will place the listing at the end of "filename" without otherwise modifying the file. Input to commands can be redirected as well. For example, "cmnd< filename" will cause "cmnd" to read from "filename" instead of the console. If it is desired to pass the output of one program to the input of another, a "pipe" for doing this is provided. For instance "cmnd1|cmnd2" will cause the output of cmnd1 to be used as input to cmnd2. These pipes can be used any number of times within a command line. For example, "c1|c2|c3|c4".

PHYSICAL DEVICE DRIVERS

The file system allows uniform treatment of both disk files and physical devices such as line printers, terminals and modems. To use this feature, simply write a program which reads a character from the device into the L register (if it reads) or outputs the content of the L register to the device (if it writes). On input, the value 0 (null) should be converted into 8000H (end-of-file) in the HL register pair. On output, the value 0DH (carriage return) should cause both 0DH and 0AH (line feed) to be sent to the device. If a device is write only (such as a line printer), it can be read protected (see chmod). If a device is read only (such as a keyboard) it can be write protected. If it can be read or written (such as a terminal) no protection bits are necessary but the routine must be informed whether it is to read or write the device. The operating system manages this by passing the value 80H in the H register if it is to read and 00H if it is to write. Thus, a simple check of the H register can be made at the beginning of the driver to decide which action it should take. Once the routine is written and an object module has been generated, check its length with "ls -l". Modules to be used in the capacity must be 256 bytes or less in length. This is plenty for most devices. Now, change the mode of the file to a physical device (see chmod) with the appropriate read/write protection. Once this is done, if output is directed to this filename or inputs are requested from it, the operating system will not return the actual content of the file. Instead, it will cause the program in the file to be loaded and run to provide the requested I/O. This is a very powerful capability of the operating system.

SHELL PROGRAMMING

Frequently, in developing programs or using the system, a user will wish to execute several lengthy commands each time an experiment is
performed. Rather than type the commands over and over again, the user can place them in a file and cause them to be executed by typing a simpler command. For instance, the command "sh<file" causes the operating shell to read from "file". This will cause the commands in "file" to be executed. In doing this, the commands themselves will be echoed to the terminal as if the user were typing them. If this is not desired, try "sh file". This will eliminate the command echo. In fact, several files can be executed in this manner if desired as in "sh file1 file2 file3". If the user wishes, the file can be declared to be an executable file (see chmod) in which case, the operating system will automatically interpret the file content as commands. Once this is done, typing "file" will execute the set of commands in the file.

ARGUMENT PASSING

When a command line is entered, the first name is the program to run and any other names are called arguments to that program. The operating system buffers these arguments and they can be accessed by any program running (see arg). Each program run is directly passed the number of arguments on the command line. Tokens which redirect I/O activity are not counted as arguments and will not be passed to the program. Thus, the program does not have to check its parameters to be sure they are really arguments and not I/O redirection.

Shell programs can be passed arguments as well. This is discussed in detail in the system call section (see system). If a command line inside a file which is declared executable and the user wishes to refer to arguments on the command line which invoked the file, the user can do so using dollar sign ($) followed by a number where the number refers to the argument order with 0 being the first argument on the line (the command name). Dollar sign followed by no number will stop and read an argument from the console and dollar sign followed immediately (no spaces or tabs) by a string in double quotes will print the string at the console before reading the argument. This allows the user to prompt for arguments.

Assembly language programs as well can receive arguments from the user through the operating system. This is done in an identical manner as any command. Assembly programs can call the arg routine to receive arguments just as well as C programs. For complete detail on passing arguments and other operating system entry points, see the system call section of this manual.

CONSOLE OPTIONS

The console represents the primary I/O device through which a user communicates with the operating system. The actual routines for reading and writing the console are placed in the bios section of the operating system (see mounting instructions). There is a file in the root directory named "con" which jumps to these routines so if any programs wish to do I/O directly to the console, they can open the "/con" file and then use getchar, getc, putchar or putc to read or write it. Several options can be controlled which relate to the console. In particular, the character delete (backspace), the line
delete and tabstops can all be set up to be any value desired (see stty). Even the prompt string which the system uses to alert the user for more input can be set using stty. The visual editor (vi) has a file for deriving the sequences it sends to the terminal for moving the cursor, etc. (see vi). This allows the system to be set up for almost any smart terminal. Finally, the console terminal should have a keyboard and display capable of displaying the full ASCII character set. This is primarily needed for C programming where various brackets and braces are used. However, the operating system does distinguish between upper and lower case itself.

SYSTEM OPERATION

This section gives a brief introduction to the internal operation of the system. This knowledge is in no way required for its use and is given only as a matter of information.

The filesystem of the operating system is simply a collection of blocks called i-nodes. Physically, these are 256 byte sectors on disks. The system refers to these with an integer between 0 and 32767. In doing this, the system becomes completely independent of the disk subsystem and is easily transported to other disk systems. The routines in the bios section are responsible for decomposing the i-node passed to them into disk/track/sector identities. This is made easy by the inclusion of a routine for producing quotient and remainder of a divide operation. I-nodes in the filesystem are used to store both file content and a forward linked list for determining what i-nodes are in use for files and what the next i-node in the file is. The linked list is distributed on the disk and can be found on every 128th i-node starting with i-node 0. Since an i-node which is used for the linked list also stores 256 bytes, it can be thought of as 129 integers (16 bit). Each integer represents the state of its corresponding i-node within that block on the filesystem, the first such integer being its own state (busy). The integer can be interpreted as four different states. If it is 0, then the corresponding i-node is idle and can be allocated for a file to utilize. If it is positive, the i-node is busy and the integer represents the next i-node in the file being accessed. If its value is -1 (0FFFFH, 0xffff), the i-node does not exist in the disk system. This can be used to prevent the operating system from accessing i-node which do not exist in the "tail" of the disk and could potentially be used to prevent access of defective media. If the integer is negative but not equal to -1, then the corresponding i-node is the last sector in the file being accessed and if lower byte of the integer is the actual number of bytes used by the file on the i-node. This linked list is initialized by the fmt command such that all i-nodes except those in the linked list itself are idle. The root directory is also set up when the media is formatted. It is empty except for the name "." (its own reference value), and the root directory always begins in i-node 1.

Beyond this, nothing in the filesystem is structured. I-nodes are allocated sequentially from the linked list and will be used in blocks if possible. This prevents slow access due to file fragmentation.
I-nodes can be used for files and directories alike. When a new directory is created, the system simply opens a new file, places "." and ".." in it (along with the starting i-nodes for each) and places the name and its starting i-node in the parent directory. This simple structure allows the creation of very sophisticated filesystem trees.
00000H - 08BFFH  UNIX commands load at location 0. This allows 35 K bytes for commands.

08C00H - 090FFH  UNIX stack area (1 K byte).

09100H - 0A0FFH  UNIX variable area and disk buffers (4 K bytes).

0A100H - 0BFFFFH This area is not used by UNIX. Users can place video RAMs or other software here and rely on it not being altered by the operating system.

0C000H - 0FFFFH  The μUNIX system kernel (16 K bytes).

To begin with, only a small amount of memory is needed at location 0, say 4 K bytes, but all other segments must be fully supplied.
The following is a list of commands provided in the /bin/ directory on your system disk. They are briefly described here and the pages that follow contain a more complete description of each. Any command can be executed by simply typing its name.

`as` - Z-80 assembler  
`asm` - 8086 assembler  
`cat` - concatenate files  
`cc` - Z-80 C compiler  
`ccc` - 8086 C compiler  
`cd` - change directories  
`chmod` - change file mode  
`cmp` - compare files  
`cp` - copy files  
`cpr` - print C programs  
`demount` - remove a mounted volume  
`du` - summarize disk usage  
`dup` - duplicate a disk  
`echo` - type arguments  
`fmt` - format a new disk  
`inode` - examine inodes  
`ld` - Z-80 link editor  
`ldr` - 8086 link editor  
`ls` - list directory contents  
`mk` - prepare a C file for execution  
`mkdir` - make a new directory  
`mount` - access a volume  
`mv` - move or rename files  
`mvdir` - move or rename directories  
`pr` - format files for printing  
`pwd` - print working directory  
`reloc` - relocate to absolute  
`rm` - remove files  
`rmdir` - remove directories  
`script` - the text processor  
`sh` - the command processor  
`ssty` - change crt parameters  
`uucp` - unix to unix copy  
`vi` - visual screen editor  
`xas` - executable Z-80 assembler  
`xcc` - executable C compiler  
`xd` - dump a file in hexadecimal  
`xld` - executable link editor  
`xsh` - executable command processor
NAME arg - get the pointer to an argument

SYNOPSIS arg(number)

DESCRIPTION ARG expects only one argument which it uses as the command argument number which is being requested. It returns a pointer to the desired argument if it exists. If not, it returns a pointer to a null. The argument being requested is found on the command line which invoked the function that called arg. Argument 0 on this line is the command name itself. I/O redirection tokens are not counted as arguments and cannot be found using arg. As an example,

cmd abc def <temp xyz>looper

will have as arguments:

0 - "cmd" 1 - "abc" 2 - "def" 3 - "xyz"

and arg will return pointers to each as requested.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO system.
NAME close - cease accessing a file

SYNOPSIS close(mode)

DESCRIPTION This routine closes the file currently being used for the standard input and/or the standard output. The argument passed to it indicates the mode of the file to be closed as indicated below.

0    Close the file being used as the standard input, resume reading the previous standard input file.
1    Close the file being used as the standard output, resume writing the previous standard output file.
2    Do both 0 and 1.

As indicated, the file previously in use as the standard input, standard output or both resumes its former status. Closing a file using mode 1 truncates all characters beyond the current write pointer. Closing with mode 2 does not. This is important when using the seek subroutine. Close returns no values.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO open, seek, getchar, putchar.
NAME compare - compare two strings in memory

SYNOPSIS compare(&s1,&s2)

DESCRIPTION COMPARE is passed two arguments which are assumed to be addresses of two strings in memory which are terminated by NULL characters (0x00). The routine compares the two and returns a value of 0 (not equal) or 1 (equal). Either string or both may contain the wildcard (*). Remember that the C compiler treats quoted strings as an address. This allows the user to program:

    if(compare(&string1,"hello"))
            goto do_hello;

for testing input text to desired responses.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO None.
NAME fprintf - print a value into a file

SYNOPSIS fprintf(id, format, value)

DESCRIPTION FPRINTF allows the user to print values inside of strings in many different fashions. The results can be directed to any file open for writing in the system. It requires exactly three arguments. The first is the fileid returned by open when the desired file was last opened for writing. The second argument is the address of a format string which controls the output. The third argument is the value to print. FPRINTF will read the format string and will simply type any characters not preceded by the percent (%) symbol directly into the file identified by id. If a percent symbol is found, subsequent characters control the printing as follow:

%Nd     Print value as a signed decimal using N character positions.
%Nu     Print value as an unsigned decimal using N character positions.
%Nx     Print value as a hexadecimal number using N character positions.
%No     Print value as an octal number using N character positions.
%Nb     Print value as a binary number using N character positions.
%c      Print value as a single character.
%s      Print the null terminated string at the address value. i.e. use value as a pointer.

Use of a length field (N) is optional. If the number cannot be printed in N positions, it will be printed anyway using as many as required. If N has a leading zero (0), the value will be printed with leading zeroes. Tabs are expanded as specified by the tabstops set by stty.

WARNINGS Never write to an invalid fileid.

BUGS None known.

FILES None.
SEE ALSO printf, scanf, stty.
NAME getc - read a single character from a file

SYNOPSIS getc(id)

DESCRIPTION GETC requires a single argument which it uses as a fileid previously opened for reading. It will return the next character in the file. Since it returns an integer, binary data can be passed through files. When end of file is encountered, getc will return the value 32768 (0x8000) for any subsequent call. Note that this cannot be confused with real data since characters are between -256 and +255.

WARNING None.

BUGS None known.

FILES None.

SEE ALSO getchar, open, close.
NAME getchar - read the standard input

SYNOPSIS getchar()

DESCRIPTION GETCHAR operates exactly like getc but does not require an argument to determine what file to read since it always reads the file last opened for reading. Like getc, it returns the value read with 32768 (0x8000) meaning end of file.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO getc, open, close.
NAME open — access files

SYNOPSIS open(&filename, mode)

DESCRIPTION This routine opens a file as the standard input, the standard output, or both. It expects two arguments. The first is the address of a null terminated file name. The second is the mode of file access desired. Three values of mode are permitted. They are:

0 — open the file for reading only
1 — open the file for writing only
2 — open the file for reading and writing

Once open, the file becomes the standard input or output or both. The previously accessed file remains open and can be accessed using getc or putc. It will become the standard input (or output) once again when the current file is closed. It is possible to access the old standard input by using the fileid of the current file - 1 and the old standard output using the fileid of the current file + 1. This allows the user to perform I/O relative to the current fileid. When opening a file for writing, the file is assigned zero length. If a file already exists, it will be truncated to zero length. This action is not taken in mode 2. If a file cannot be accessed, open returns the value -1 which is not a valid fileid. Modes 0 and 1 return the value of the fileid which can be used with getc, putc, and printf respectively for accessing the file. Mode 2 returns a composite id equal to 16 * input_id + output_id. Directory names must end in "/" to be opened.

WARNINGS Users should not modify directories.

BUGS None known.

FILES None.

SEE ALSO close.
NAME printf - print a value at the standard output

SYNOPSIS printf(format,value)

DESCRIPTION PRINTF is identical to fprintf except that it does not accept a fileid argument. In lieu of this, it always directs its output to the file currently open as the standard output.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO fprintf.
NAME putc - write a character to a file

SYNOPSIS putc(id,byte)

DESCRIPTION PUTC expects two arguments. The first is the fileid of a previously opened file to write. The second is the character to be placed in the file. There are no constraints on the byte being written. This allows transfers of binary data to files.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO open, close, putchar.
NAME putchar - write a character to a file

SYNOPSIS putchar(byte)

DESCRIPTION PUTCHAR sends the byte passed it as an argument to the file currently open as the standard output. No constraints are made on the byte being passed. This allows binary data to be transferred.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO open, close, putc.
READ

NAME read - transfer a block from a file to memory

SYNOPSIS read(id,address,nbytes)

DESCRIPTION READ requires three arguments. The first is the fileid to be used for the transfer. The second is the address in memory where the block is to be placed. The third is the number of bytes to transfer. When finished, read will return the actual number of characters transferred. This routine can be alternated with getc or getchar on the same file and each will read sequentially. Read is most efficient when used to transfer large blocks of data but any length is permissible. The third argument, nbytes, is treated as an unsigned value. This permits moves of up to 65535 characters in a single block. If the file does not contain as many characters as requested, read will transfer all that remain and return that number.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO open, close, getc, getchar, write.
NAME scanf - text and number reading routine

SYNOPSIS scanf(&bptr,&format,&variable)

DESCRIPTION SCANNF expects three arguments. The first is the address of a
pointer which CONTAINS the address of the string to process. The
pointer would normally be a character pointer previously initialized
to the address of a buffer loaded with ASCII text. The second argument
is the address of a format string which will control scanf in matching
text and reading values. The third argument should be a character,
an integer or a character pointer depending upon the intended value to be
transferred. Scanf will compare the text string to the format string.
If at any time they do not match, scanf will terminate and return the
value 0 and they buffer pointer variable will not be moved. If they
match to the end of the format string (a null terminates it), scanf
will return the value 1 and the buffer pointer will be moved so that
it points to the next character in the buffer. While matching, scanf
recognizes special sequences in the format string. These are given
below.

*                  Matches any number of blanks or tabs in the text string.
%d                Matches a decimal number. The value of the number will be
                  placed in the variable.
%x                Matches a hexadecimal number. The variable will be set equal
to the number.
%o                Matches an octal number. The variable is set to its value.
%n                Matches a number specifying its own base. Such numbers are
                  4096 (decimal), 0x1000 (hexadecimal), 01234 (octal) and 0b0101
                  (binary). This is probably the most convenient format.
%c                Reads a single character from the text and places it in the
                  variable. In this case, the variable should be declared as a
                  character.
%s                Reads a string from the text into the memory pointed to by the
                  variable. In this case, the variable should be a character
                  pointer and must have been initialized with a buffer address.

WARNINGS None.
SCANS

BUGS None known.

FILES None.

SEE ALSO compare.
NAME seek — move file pointers

SYNOPSIS seek(page, byte, mode)

DESCRIPTION SEEK expects three arguments. The first is the page on which the file pointer is to be positioned where a page is interpreted as 256 bytes. The second is the specific byte within the page to which the file pointer is to point. The combination of these is used to move the file pointer where the final position is taken to be 256 * page + byte. It is not necessary for byte to be in the range 0 to 255. For example, page=0, byte=1024 is equivalent to page=4, byte=0. The third argument determines which file pointer to move. Mode is decoded as:

0 — read pointer
1 — write pointer
2 — both read and write pointers

Notice that no fileid is passed since seek always moves the standard input and/or the standard output pointers. After seek is completed, getchar, getc, putchar,putc, as well as read and write will begin accessing the file beginning at the new pointer values.

WARNINGS If the files open are physical devices, seek merely returns. It is not possible to backup a physical device.

BUGS None known.

FILES None.

SEE ALSO open, close.
NAME string - read a string into memory

SYNOPSIS string(&buffer)

DESCRIPTION STRING expects a single argument which it uses as the
description of a buffer in which it loads a string. Characters making up
the string are read from the standard input until a newline is found.
When this occurs, string returns the number of characters read from
the input. The routine also places a null after the received newline
so that the string can be processed by other routines but the null
does not count in the count returned by it. While reading the standard
input, characters are echoed to the standard output. It recognizes the
character delete, line delete and tabstops set by stty. Tabs are
converted to blanks only in the echoed output and the true tab value
(0x09) is stored in the buffer. After the string is read, the routine
checks to see if the first character is the exclamation point (!). If
it is, string calls the operating system on the remainder of the
buffer and then reads another string from the standard input. This
allows easy access to the operating system from new programs being
built.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO stty.
NAME system - execute a command

SYNOPSIS system(&buffer)

DESCRIPTION SYSTEM is the true command processor in the UNIX operating system. It expects a single argument which it uses as the address of a null terminated string containing commands. It breaks the tokens found in the command string into separate strings. The first token which is not an I/O redirect control is taken to be the command. The program having that name will be loaded into memory and executed as an object module. The program must be in relocatable format as generated by the link editor and will be relocated as required by the operating system. The system will search directories for the command name as specified by the path variable. The path variable can be modified using stty. Normally, it is:

"/bin/::"

which will cause the system to search first in the /bin directory and then second in the current directory. The colons denote the end of the path to which the command name is appended. In other words, /bin/:: is the path and the command xyz is typed, the following paths will be used for finding xyz:

/bin/xyz

All other tokens found on the line are considered to be arguments to the command. Any argument can be referenced by the user using the arg function. When the command is actually loaded, it is passed a single argument which is the number of arguments which was present on the command line invoking the program named by the command. Tokens are separated by spaces, tabs, or I/O redirect symbols. Redirect refers to the operating system ability to establish what files are the standard input or standard output. This is done by preceding a name with the less than symbol (<) for the standard input or the greater than symbol (>) for the standard output. Two greater than symbols (>>>) specify appending to the named file as the standard output. Some examples follow:

ls>file cmd<abc>/xyz ls /bin/ >>file

When specifying I/O redirect, the name must immediately follow the redirect symbol. Pseudo pipes are supported by system. This allows output from one command to be run as input to the next command. For example,

ls>pr>/lpr
will cause the output from ls to be placed as input to pr whose output in turn is directed to /lpr.

Multiple commands may be present in the buffer provided they are separated by semicolons (;).

The arguments of the calling function can be used within the current function. Dollar sign ($) specifies this action if it is followed immediately by the argument number desired. For example, a shell program xyz contains the statement:

```
pr $1>$2
```

When invoked by the statement xyz abc temp, the example will become,

```
pr abc>temp
```

If dollar sign is not followed by a number, system will read a string to use as the argument from /con. If dollar sign is immediately followed by a string in double quotes, the string will be printed at /con and then the argument will be read from /con. It is possible to concatenate arguments together by simply not placing spaces between them. For instance,

```
$1.c $1$2 "$Enter name".s
```

are all proper values. The name of the calling command can be accessed using $0.

System does not break up strings in double quotes but it does translate backslash sequences within them. For instance,

```
"hello there!\\n"
```

will be kept intact with only the newline (\n) being translated.

When system is finished processing the entire command string, it will return the same value returned to it by the last command executed.

**WARNINGS** None.

**BUGS** Double quoted strings cannot be used as I/O redirect token names (ls > "Hello there").

**FILES** None.

**SEE ALSO** arg, stty.
NAME write - transfer a block from memory to a file

SYNOPSIS write(id, address, nbytes)

DESCRIPTION WRITE requires three arguments. The first is the fileid to be used for the transfer. The second is the address in memory where the block is to be found. The third is the number of bytes to transfer. When finished, write will return the actual number of characters transferred. This routine can be alternated with putc or putchar on the same file and each will write sequentially. Write is most efficient when used to transfer large blocks of data but any length is permissible. The third argument, nbytes, is treated as an unsigned value. This permits moves of up to 65535 characters in a single block.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO open, close, put, putchar, read.
NAME unix - start the operating system

SYNOPSIS unix(start)

DESCRIPTION UNIX is the beginning entry point for the entire operating system. It expects a single argument. The value of this argument determines the action taken. If the value is 1, the operating system makes a cold start. This implies:

The tabs are set to 4. The erase is set to backspace (0x08). The prompt is set to "%". The path is set to "/bin:". Any file I/O is terminated, no files are open. The argument pointer is reset. The current directory is set to the root (/). If a file named profile exists in the root, it will be executed as shell commands. The shell is executed using /con for I/O.

If the value is 0, the operating system makes a warm start. This implies:

Any files open will be closed gracefully. The argument pointer is reset. The shell is executed using /con for I/O.

If the value is 2, the operating system initializes the file system and returns to the caller. This allows the user to use the file system for transferring files from other operating systems.

WARNINGS If sh or /con are missing, the system cannot boot.

BUGS None known.

FILES None.

SEE ALSO None.
NAME xas - the real Z-80 assembler

SYNOPSIS xas [-l] [-oname] [-p#] [-x] file_name

DESCRIPTION XAS accepts a number of argument flags. These are specified below.

-1 Generate a listing
-oname Place the output in the "name" file
-p# Page size (for listings) is # lines. (the default is 66).
-x Do not generate relocatable code.

The final argument is the filename to assemble. The input language to
the assembler must be in upper case and is a derivative of the
Technical Design Labs (TDL) modifications of the 8080 programming
language to include Z-80 extensions.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO as, cc, ld.
NAME xcc - executable C compiler

SYNOPSIS cc source destination

DESCRIPTION XCC is the real C compiler. It accepts two arguments. The first is assumed to be a C program to compile and the second is used as the destination file name for the resulting assembly language program. For example,

```
xcc file.c result.s
```

will place an assembly language program in the file result.s. This compiler accepts standard C program text with some exceptions. It does not allow for multiple dimensional arrays (one-dimensional only), it does not support structures in any way, and it does not allow initialized variables. It also does not support the #define statement. It does support all other standard features of the C language.

WARNINGS Unlike cc, xcc does not append the ".c" and ".s" suffixes so remember to enter them if they are required.

BUGS None known.

FILES None.

SEE ALSO cc.
NAME xd - hexadecimal dump of a file.

SYNOPSIS xd file

DESCRIPTION XD prints the named file (if found) in hexadecimal notation. The address, beginning with 0, is printed next to each line of 16 bytes. XD can only be terminated by cnt1-Q.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO None.
NAME xld - the real link editor


DESCRIPTION XLD accepts several arguments which are described below.
-1 Produce a listing at the console
-1name Produce a listing at "name"
-oname Send the resulting module to "name"
-r# Offset the relocatable code by #
-p# Page size (for listings) is # lines
-e Retain any ENTRY definitions in the files (i.e. place them in the output)

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO ld, as, cc.
NAME xsh - executable shell

SYNOPSIS Never invoke this command

DESCRIPTION XSH is used by the operating system for executing files which are deemed shell programs. This is determined by the mode byte of the file and is checked by the operating system when a command is invoked.

WARNINGS Users should not invoke XSH or in any manner modify it. It is included here only for completeness.

BUGS None known.

FILES None.

SEE ALSO None.
NAME vi - visual editor

SYNOPSIS vi file_name

DESCRIPTION This manual describes the commands and capabilities of the VI screen oriented text editor. For the duration of this manual, the character ^ should be interpreted as the word "control". For instance, ^F means control-F. The symbol <cr> means carriage return, the symbol <sp> means the space character, the symbol <bs> means the backspace character, and <num> means an optional number. All other characters should be interpreted exactly as depicted and be sure to observe their case (upper or lower) as it is significant.

VISUAL MODE COMMANDS

I. Vertical Motion Group

<num>- Move up num lines in the file, or to top.
<num><cr> Move down num lines in the file, or to end.
<num>g or G Go to line number num in the file, or to end if num is greater than total number of lines in file.
h or H Go to top line currently displayed on screen.
l or L Go to bottom line currently displayed on screen.
^F Scroll down 1 screen size in the file.
^B Scroll back 1 screen size in the file.
^D Scroll down 1/2 screen size in the file.
^U Scroll back 1/2 screen size in the file.

II. Display Control Group

z<cr> Redraw screen, with cursor on top line.
z. Redraw screen, with cursor on center line.
Redraw screen, with cursor on bottom line.

Redraw screen.

III. Locate Group

/string<cr> Go to next occurrence of character string "string" in the file, searching in the forward direction. If end of file is hit, search will abort.

?string<cr> Go to next occurrence of "string" in the file, searching backwards. If top of file is hit, search will abort.

n Go to next occurrence of the last locate string entered. Search in same direction as before.

N Go to next occurrence of the last locate string entered. Reverse direction of search.

% When the cursor is currently pointing to a brace, bracket, or parenthesis, this command will locate the matching brace, bracket, or parenthesis. If not found, terminal will "beep".

IV. Horizontal Motion Group

<num><sp> Space forward num characters on current line, or to end of line.

<num><bs> Back up num characters on current line, or to start of line.

^ Go to first character in current line.

^ Go to first non-blank character in current line.

$ Go to end of current line.

<num>w Move forward to start of numth following alpha-numeric word on current line, or to end of line.

<num>e Move forward to end of numth following alpha-numeric word.

<num>b Move back to start of numth previous alpha-numeric word.

<num>w Move forward to start of numth following blank delimited word on current line, or to end of line.
VI

<num>E Move forward to end of numth following blank delimited word.
<num>B Move back to start of numth previous blank delimited word.

V. Delete Group

<num>dd Delete num lines, starting with the current line, or delete to end of file.
<num>d<sp> Delete the next num characters on the current line, starting with the current character, or to end of line.
<num>x Same as <num>d<sp>, only shorter.
<sp>d$ Delete from cursor to end of current line.
<num>dw Delete from cursor to start of numth following alpha-numeric word on current line.
<num>de Delete from cursor to end of numth following alpha-numeric word on current line.
<num>dW Delete from cursor to start of numth following space delimited word on current line.
<num>dE Delete from cursor to end of numth following space delimited word on current line.

VI. Yank Group

<num>yy Yank num lines, starting with the current line, or yank to end of file.
<num>y<sp> Yank the next num characters on the current line, starting with the current character, or to end of line.
<sp>y$ Yank from cursor to end of current line.
<num>yw Yank from cursor to start of numth following alpha-numeric word on current line.
<num>ye Yank from cursor to end of numth following alpha-numeric word on current line.
<num>yW Yank from cursor to start of numth following space delimited word on current line.
delimited word on current line.

\texttt{<num>\textasciitilde E}  Yank from cursor to end of numth following space delimited word on current line.

VII. Change Group

\texttt{<num>cc}  Change num lines, starting with the current line.

\texttt{<num>c<sp>}  Change the next num characters on the current line, starting with the current character, or to end of line.

\texttt{c\$}  Change from cursor to end of current line.

\texttt{<num>cw}  Change from cursor to start of numth following alpha-numeric word on current line.

\texttt{<num>ce}  Change from cursor to end of numth following alpha-numeric word on current line.

\texttt{<num>cW}  Change from cursor to start of numth following space delimited word on current line.

\texttt{<num>cE}  Change from cursor to end of numth following space delimited word on current line.

VIII. Input Group

\texttt{i}  Insert new text between current character and previous character.

\texttt{a}  Append new text between current character and next character.

\texttt{(ESCAPE)}  Stop inputing characters into the file.

\texttt{r}  Replace the current character with the next character typed in. New line characters cannot be changed with this command.

\texttt{o}  Open a new line below the current line, and start inputting on this line.

\texttt{O}  Open a new line above the current line, and start inputting on this line.

\texttt{p}  Put the contents of the yank buffer between the current character and the next character (or between the current...
line and the next line, if the yank buffer contains lines).

P
Put the contents of the yank buffer between the current character and the previous character (or between the current line and the previous line, if the yank buffer contains lines).

IX. Macro Group

Macros are allowed to call each other, or themselves. Any command can be used in a macro. Any error which causes a "beep" terminates a macro.

<num>m
Execute the current definition of macro num (1 to 4).

<num>M
Display the current definition of macro num on the command line.

<num>s or S
Enter a new definition for macro num on the command line.

X. Miscellaneous Commands

^G
Print file statistics on command line.

<num>t
Set tabstops to every numth character.

D
Display the current contents of the yank buffer.

j
Join the current line and the line below it into one line, deleting the newline between them.

C
Toggle the caps lock option (converts all letters to capitals while in input mode).

COMMAND MODE COMMANDS

I. File Manipulation Group

:w<cr>
Write file to current file name displayed with ^G command.

:w fname<cr>
Write file to file name fname.

:W<cr>
Same effect and options available as for :w<cr> above,
except that a :q<cr> will be executed after the write.

:r<cr>  Restore current file to version currently saved on disk.

:e fname<cr>  Discard current file being edited, and edit a new file
named fname, if the current edited version of the file
matches the version saved on disk.

:E fname<cr>  Discard current file being edited, and edit a new file
named fname, regardless of the state of the version of
the file saved on disk.

:y fname<cr>  Place the contents of file fname into the yank buffer.

II. Program Control Group

:q<cr>  Quit the edit session, if the current edited version of
the file matches the version saved on disk.

:Q<cr>  Quit the edit session, regardless of the state of the
version of the file saved on disk.

:!string<cr>  Execute the character string "string" as a shell
command.

USING DIFFERENT TERMINALS

Each time VI begins running, it reads a terminal profile from the file
/bin/vi.crt which contains the various sequences which cause the
terminal to perform operations on its display. The file contents are
described below.

Lines    This is a binary character which is the number of lines
available for display on the terminal screen.

Home clear<cr> This string must contain the sequence which homes and
clears the terminal screen.

Clear to end<cr> This string must contain the sequence which causes the
terminal to clear the screen from the current cursor
location to the end of line.

Insert line<cr> This string must contain the sequence which causes the
terminal to insert an empty line at the cursor location.

Delete line<cr> This string must contain the sequence which causes the
terminal to delete the line at the cursor location.

Enter insert mode: This string must contain the sequence which
causes the terminal to begin insertion of characters
received at the cursor location.

Exit insert mode: This string must contain the sequence which causes
the terminal to cease insertion of characters received
at the cursor location.

Delete character: This string must contain the sequence which causes
the terminal to erase the character at the cursor location.

row offset+16 This is a binary character which will specifies how much
offset to add to the row addresses passed to the
terminal. Normally, this value is made to be 16 which
results in no offset.

move row: This is the first part of the string which will cause
the terminal to move its cursor to a new row and column.
This portion specifies the row address as well as any
characters to send before it is sent. The row address
will be sent when the specification %d or %c is found.
If %d is used, the value is sent in ASCII decimal. If
the format is %c, it will be sent as a binary character.

col offset+16 This is a binary character which will specifies how much
offset to add to the column addresses passed to the
terminal. Normally, this value is made to be 16 which
results in no offset.

move col: This is the second part of the string which will cause
the terminal to move its cursor to a new row and column.
This portion specifies the column address as well as any
characters to send before or after it is sent. Like the
row address, the value will be inserted whenever %d or
%c is found.

Fortunately, this file can be created very simply using the "echo"
command of the operating system. First, write down the sequences and
values to send. Then, determine what ASCII characters have these
values and write them down. In doing this, control characters should
be preceeded by a backslash (\) and remember that carriage return can
be coded as \n. For instance, the sequence ESC & cntl-x can be written
as "\[\&\x]. Once these sequences are written down, they can be placed
into a file by typing:

echo "the entire sequence">filename
An example sequence, which will condition VI for a Hewlett-Packard HP2621 is shown below. Notice that the sequence has been broken in two parts for convenience.

```bash
echo "\X\C\J\n\K\n\L\n\M\n\O\n\R\n\P\n">hp.crt
echo "\P\&a%dR\n\P%dC\n">>hp.crt
```

Note also that the second piece of the string is appended using the I/O redirect append operator (>>) supported by the operating system.

The content of the file can be viewed using the "x,d" command to be sure it is correctly built. Then, move the file into the "bin" directory by typing:

```
mv filename /bin/vi.crt
```

Now, VI should be compatible with your terminal.

WARNINGS None.

BUGS None known.

FILES /bin/vi.crt

SEE ALSO None.
NAME script - text processor

SYNOPSIS script file_name

DESCRIPTION This manual describes a text formatter program entitled "MICROSCRIPT". MICROSCRIPT accepts a single argument which is the text file which is to be processed. It accepts no other arguments or flags.

The MICROSCRIPT program has been modelled roughly after the "SCRIPT" text formatter program marketed by the University of Waterloo. Extensive simplifications to the features have been made, however, to allow efficient implementation on a small microcomputer such as the Z-80. Specifically, any sort of commands which would require extensive amounts of internal buffering have been eliminated. The primary losses thus incurred are as follows:

1. Multiple column output is not supported in any way. All output must be in the form of one column per page.

2. No specific support of footnotes is provided.

3. No command is supported which requires that text be formatted first, and then printed later. For example, no support of "conditional paging" to force a block of text to all appear on the same page is provided.

4. In addition, several little used commands have been eliminated and/or changed to allow greater generality of use at the cost of a certain amount of convenience.

Despite these limitations, MICROSCRIPT is quite capable of supporting a wide range of document styles. Most of the unsupported features can be accomplished by performing a trial format of the document, and then making a few minor changes to the definition to allow for exact page placement. Only multiple column output remains impossible.
PAGE FORMAT:

All documents prepared with MICROSCRIPT are, of course, printed on pages. Every aspect of page size and characteristics are user definable within the framework of MICROSCRIPT. Each page produced by MICROSCRIPT has the following basic format:

```
| .pl |--< .lm |<-------- .ll --------> |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
```

.pl = page length
.tm = top margin
.bm = bottom margin
.lm = left margin
.ll = line length
.in = indent amount within .ll
.tt = top title line
.bt = bottom title line
ENTRY OF COMMANDS AND ARGUMENTS:

All commands to MICROSCRIPT are of the form \texttt{.\textit{xx}}, with the \texttt{.}\ beginning in column 1 of an input record, and \texttt{\textit{xx}} being a 1 or 2 letter command. Command names have been picked as much as possible to reflect their function. Arguments are optional, depending on the command, and come in two forms: numerical values, and strings. A numerical argument may be specified in any one of four ways:

\begin{itemize}
  \item $n$ \quad The numerical value $n$ is assigned to the appropriate parameter.
  \item $+n$ \quad The parameter is incremented by the value of $n$.
  \item $-n$ \quad The parameter is decremented by the value of $n$.
  \item no arg. \quad If the argument is left off entirely, the parameter returns to it's default value.
\end{itemize}

A string argument is any sequence of printable characters surrounded by two "delimiter" characters. The delimiter character can be any printable character other than space, tab, or \texttt{'$'}$. Examples of strings are as follows:

\begin{itemize}
  \item /this is a string/
  \item "\so is this"
\end{itemize}

In some commands, more than one string must be specified at once, with consecutive delimiters between each. For example:

\begin{verbatim}
/string1/string2/string3/
\end{verbatim}

You can put more than one command on one line, if desired, by following the first command with a \texttt{;'}, and then putting the leading \texttt{'} of the next command immediately after the \texttt{;'}. It is also possible to follow a command with a \texttt{;' } and then normal text to format all on one line. For example, if you wish to center just one line, it is allowable to type:

\begin{verbatim}
.ce 1;Line to center
\end{verbatim}

Finally, if some command \texttt{.xx} is not recognized, MICROSCRIPT will search on disk for a file with the name \texttt{xx}. (Note that the leading \texttt{.'} is not part of the file name.) If the file exists, MICROSCRIPT will process any commands and input lines in the specified file until it is exhausted, and will then return to the original file being processed. In this way, "macro" commands can be written for often used sequences. Nesting of macros is allowed, and is limited only by the number of buffers in the operating system.
THE FORMATTING PROCESS:

The MICROSCRIPT text formatter program operates in one of three basic modes, selected by the user. A description of the functional characteristics of each follows:

mode 1: The simplest mode of operation is "unformatted" mode, specified by the ".nf" or "no-fill" command. In this mode, lines of text from the input file are copied to the output file exactly as they appear in the input file. All system parameters listed in the PAGE FORMAT section of this document are obeyed while in this mode, except for line length (.11). In this case, lines will be whatever length you make them in the input file. This mode is primarily useful for entering tables, diagrams, and other things where the exact placement of all characters must be undisturbed. For example, the "picture" of a typical page which appears earlier in this manual was formatted using this mode.

mode 2: Both other modes operate in what is known as "fill" mode, specified by the ".fi" command. When in fill mode, words of text are copied from the input file to a separate output buffer within MICROSCRIPT, one at a time, with one space between each word. Any extra blanks between words in the input file are deleted, and line boundaries within the input file have no real significance (except for delimiting command lines). Whenever the output reaches a point where no more words can be included without exceeding the line length (.11) parameter, the buffer is printed, emptied, and the process starts over again.

mode 3: The last mode available is "fill and adjust" mode, obtained by specifying both the ".fi" and ".ad" commands. Adjust mode can be turned off by specifying ".na". If ".nf" is specified, the state of adjust mode is unimportant. Fill and adjust mode operates in exactly the same fashion as fill mode alone, except that one additional processing step takes place. Whenever it has been determined that no more words will fit into the output buffer, extra spaces are inserted between words already in the buffer until the length of the line to print is exactly equal to the line length. This document was prepared using this mode. Also, this mode is the default condition when MICROSCRIPT is first called.

As long as at least two full words can fit on any one line (of length .11), MICROSCRIPT will never hyphenate. If this condition is not met, MICROSCRIPT will, if necessary, hyphenate words to make them fit within the specified line length. Since MICROSCRIPT is not overly intelligent about it's placement of hyphens, it is best to avoid letting the line length get this small.
Whenever (in fill mode) it becomes necessary to print the output buffer and start filling it over again, this condition is referred to as a "break". The most common reason for a break to occur is that the output buffer is full. However, there are several other conditions which may occur which will cause a break even if the buffer is not full. For example, if a new paragraph is started, the last line of the old paragraph must be printed before the new paragraph can begin, even if it is not full. Also, many other commands such as ".sp" (skip spaces) and ".bp" (begin new page) will cause a break. Whenever a break does occur, the last line printed will not be adjusted before printing, even if ".ad" is specified. This is necessary, since the output buffer is not really full.

UNDERSCORING:

In order to underscore text within MICROSCRIPT, it is only necessary to surround the text to underscore with "\" characters. For example, if the following line is entered:

\Every good boy does fine\.

The following will be the result:

Every good boy does fine.

If in no-fill mode, everything between the "\" characters will be underscored, including any blanks. If in fill mode, only non-blank characters can be underscored.

If it is really desired to enter a "\" into the text stream, you can type "\\". Also, if some line contains only one "\", everything from the "\" to the end of the line will be underscored.

LIST OF AVAILABLE COMMANDS:

The remainder of this manual consists of a list of all currently available commands within MICROSCRIPT, and a short description of the function of each. Whenever an underscored quantity appears in a list of argument options, this quantity is the "default" value, obtained if the argument is omitted entirely. Also, if a "*" appears before the command name, this indicates that the command causes a break.
.ad  Turn on "adjust" mode. Note that this command will do nothing if ".fi" is not specified.

.bm <n!+n!-n>  Set bottom of page margin to <argument> lines.

* .bp <n!+n!+n!-n>  Terminate current page, and begin a new page with page number <argument>. Default is the next page number. +n and -n argument values are relative off of the current page number (not the next page number).

* .br  Initiate a "break". (Refer to section on the formatting process for a definition of break.)

.bt <n!+n!-n> /string1/string2/string3/  Bottom title line definition (see section on page format for placement on page). The bottom title line will be printed as the nth line of the bottom margin (n = <argument>). The line will consist of string1 left adjusted, string2 centered, and string3 right adjusted within the line length parameter. If the sequence "$$" appears in any string, it will be replaced by the current page number printed as 2 decimal digits. Similarly, the sequence "$$" will print the current page number as 3 digits. Note that these translations only apply to title line definitions. MICROSCRIPT defaults to " .bt 0 //\". If the value of <argument> does not fall within the range of the bottom margin, the title line is not printed.

* .ce <until ..ec;n>  Center the next <argument> lines in the input file. If no argument is given, all further lines will be centered until a ".ec" command is encountered.

.ds  From the next line printed until some other command overrides, start double spacing between all printed lines of text.

.ec  Terminate line centering mode. If the mode was already off, this command will do nothing.

.fi  Turn on "fill" mode.

* .in <n!+n!+n!-n>  Indent all following printed lines by <argument> spaces, until some other command overrides. If in fill mode, the "effective" available line length will be decreased by the indent amount, so that the overall line length will remain the same.
* .1e

Terminate a list item. This is accomplished by setting the indent amount to 0, and then executing a "sp 1". (See the ".li" command below for the definition of a list item.)

* .li <n!n!n!-n> /string/  Start a list item. A list item consists of a string (the /string/ argument), and a block of text which is associated with that string. The .li command will execute a "sp 1", and then set the indent amount to <argument>. Note that the same variable is affected here as in the ".in" command. All lines printed after the .li command will be indented by <argument> spaces. In addition, the first line printed after the .li command is issued will have the contents of /string/ printed on that line, located at the very beginning of the line (without any indent). For example, the command list that you are now reading was formatted with .li commands, with <argument> = 21, and with each /string/ = to the name and argument list for the command. If /string/ is too big to fit within the allotted space provided by the indent, the "effective" line length will be decreased accordingly for that first line only. After the first line is printed, the indent will revert to the value specified in the .li command. Note that the .li command description that you are now reading is an example of this situation.

* .11 <@n!n!n!-n>  Set current line length to <argument> characters. This value is only used in fill mode. Note that .in, .li, and .p may all cause the "effective" value of this variable to decreased.

*.lm <@n!n!n!-n>  Set left margin on page to <argument> characters. Note that this variable does not affect the defined line length.

.na

Turn off "adjust" mode.

*.nf

Turn off "fill" mode.

*.p

Begin a new paragraph. This command executes a "sp 1", and then increments the indent amount by <pi> characters. This extra indent amount applies only to the first line printed after the .p command. After the first line, the indent amount reverts to it's previous value. The value of <pi> can be set with the ".pi" command.

* .pi <@n!n!n!-n>  Set paragraph indent amount to <argument>.
.pl <argument> Set current page length to <argument> lines per page.

* .sp <argument>
Skip <argument> lines in the output file. Note that if ".ds" is specified, 2*<argument> lines will actually be skipped.

.ss
From the next line printed until some other command overrides, start single spacing all printed lines of text.

.tm <argument>
Set top of page margin to <argument> lines.

.tt <argument> /string1/string2/string3/ Top title line definition. See the .bt (Bottom title line) description for details. This command operates in the same way as .bt, except that it is associated with the top margin.

* <tab>
If a line of text begins with a tab character, MICROSCRIPT will perform the same actions incurred by issuing a ".p" command, except that it does not print the blank line between paragraphs. When in the unformatted mode, no action at all is taken.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO None.
NAME asm - the 8086 assembler

SYNOPSIS asm [-ls] filename [filename] ....

DESCRIPTION ASM is the 8086 assembler. It accepts one or more
arguments which it treats as the path to a file to assemble. Filenames
given in the arguments to asm are appended with ".s" prior to
searching. The wild card character (*) is permitted and asm will
assemble all files ending in ".s" which match the name. Asm accepts
two flags, if desired, which can produce a listing and/or a symbol
table.

-l Produce a listing
-s Produce a symbol table

In both cases, the output will appear at the console unless the
standard output has been redirected to a file. An example of this
follows.

    asm -l /temp/* >temp.listing

The assembler accepts lower case files containing standard Intel
mnemonics for the 8086.

When the assembler is finished, there will be a ".o" file for each
".s" file which it was asked to assemble. These files are ready for
link editing.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO ccc, ldr.
NAME ccc - the 8086 C compiler

SYNOPSIS ccc filename [filename] ....

DESCRIPTION CCC is the 8086 C compiler. It accepts any number of arguments greater than one and treats each as a pathname to a ".c" file to be compiled. The wild card character (*) is permitted and the compiler will compile all ".c" files which match the specified name. For example,

    ccc xyzzy /temp/ab*

will compile "xyzzy.c" and all files in the directory "temp" whose names begin with "ab" and end in ".c".

When the compiler is finished, there will be a ".s" file for each ".c" file which was compiled. These files are standard 8086 assembly language programs and are ready to be assembled by the 8086 assembler.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO asm, ldr.
NAME  ldr - the 8086 link editor

SYNOPSIS  ldr [-ls] filename [filename] ....

DESCRIPTION LDR is the 8086 link editor. It accepts any number of arguments greater than one which it treats as a path to a ".o" file to be link edited. The wild card character (*) can be used in which case the link editor will link all matching files ending in ".o". It accepts two flags.

-1 Produce a listing of the symbol table. The listing will appear at the standard output and can be re-directed to a file or printer.

-s Place a copy of the symbol table in the output. This is useful for debuggers.

When the link editor has finished, there will be a single file created which will have the name of the first argument but without the ".o". This file is the executable module.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO ccc, asm..
NAME  cpr - print C programs

SYNOPSIS  cpr [ -f ] [ -i# ] [ -p# ] filename

DESCRIPTION  CPR is a useful utility for printing C source programs. It provides pagination, titles, statement numbering, as well as level and bracket nesting. It accepts a single argument which is the filename to print. It also accepts a set of flags for setting options.

- f        Skip to a new page at the end of each C function.
- i#       Set page indent amount to # (default is 0).
- p#       Set page length to # (default is 66).
- f i# p#  If all options are requested.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO None.
NAME as - Z-80 assembler

SYNOPSIS as as_pgm

DESCRIPTION AS accepts a single argument which it treats as the name of a ".s" file. On completion, a new file named as_pgm.o will exist in the current directory. This file is a load module program ready to be link edited. For example,

    as myfile

will cause "myfile.s" to be assembled producing "myfile.o". Note that the user should not enter the ".s" suffix. The source file will not be modified in any manner. For more details on the Z-80 assembly language, see the xas manual. Listings cannot be produced using this command. If a listing is desired, use xas.

WARNINGS This routine is simply a shell program which calls the real assembler, xas.

BUGS None known.

FILES None.

SEE ALSO mk, cc, xas, ld.
NAME cat - concatenate files

SYNOPSIS cat [file_name] ........

DESCRIPTION The cat command permits the user to type those files named at the standard output. It accepts any number of arguments including none. If none are given, it simply echoes the standard input to the standard output until the end of file is reached. If arguments are passed, all files named are typed. The wild card (*) is permitted and cat will type all files with matching names. Some examples are given below.

    cat *.c bios.s
    cat ../tempfile
    cat /user/help/* manual/*

This command requires no flags. Tabs are always translated as specified by the stty setting. (See also STTY)

WARNINGS Binary files (non-ASCII) cannot be viewed by cat. Attempts to do so may cause system buffers to be overflowed.

BUGS None known.

FILES None.

SEE ALSO stty.
NAME cc - C compiler

SYNOPSIS cc c_pgm

DESCRIPTION CC accepts a single argument which it treats as the name of a "c" file. On completion, a new file named c_pgm.s will exist in the current directory. This file is an assembly language program ready to be assembled. For example,

   cc myfile

will cause "myfile.c" to be compiled producing "myfile.s". Not that the user should not enter the "c" suffix. The source file will not be modified in any manner. For more details on the C programming language, see the xcc manual.

WARNINGS This routine is simply a shell program which calls the real compiler, xcc.

BUGS None known.

FILES None.

SEE ALSO mk, as, xcc, ld.
NAME cd - change directories

SYNOPSIS cd [path_name]

DESCRIPTION The cd command permits motion within the file hierarchy. It accepts either no arguments or a single argument. If no arguments are given, it moves to the root directory. If an argument is passed, it moves to that directory. The argument may begin with a slash in which case the search begins at the root. If there is no leading slash, it begins with the current directory. In either case, the argument should be a path to a valid directory. If any fault is found with the path, an explanatory message is printed. Some examples follow.

cd cd ../temp/abc cd /user/bin

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO mkdir, rmdir, mvdir
NAME chmod - change mode of files

SYNOPSIS chmod newmode filename [filenames]

DESCRIPTION The chmod command allows the user to modify the mode word associated with a file. It accepts as arguments a new mode value and a list of filenames. The filename list can use the wild card (*) and can be paths to other directories. The new mode value is given in symbolic form using combinations of the letters shown below.

1 No list. The filename will not be listed by the ls command unless the -a option (of ls) is used.

x Shell executable. The file will be interpreted as shell commands when entered as a command rather than be loaded and executed directly.

p Physical device driver. If input or output is directed to such a file, the file will be used as a program for doing I/O. The length of such a file must be less than 256 bytes.

w Write protected. Files protected in this manner cannot be written. This applies to physical device drivers as well.

r Read protected. Files protected in this manner cannot be read. This is useful for physical device drivers which are write only. (ie. line printers).

o No modes. This removes all protection and/or mode bits so the file can be manipulated normally.

Any unspecified option is disabled in the resulting mode. When specifying more than one flag, the flags must be ordered as listed (1xpwr). Directories can only be modified by the no list option and ordinary files cannot be made into directories by chmod. Some examples follow.

chmod lpw modem /dev/* chmod l /bin/

Note that when directories are named, the name must end in a forward slash.

Be careful with the use of the wild card (*) and this command. If the modes of physical device drivers are made wrong, the file will be
CHMOD

destroyed if written and must be remade. If the console driver (/con)
is damaged in this manner, the disk becomes useable.

BUGS None known.

FILES None.

SEE ALSO ls
NAME cmp - compare two files

SYNOPSIS cmp file1 file2

DESCRIPTION CMP does file comparison on a character by character basis. It accepts two arguments which are taken to be filenames. If either cannot be found, a message is printed. Otherwise, the two are compared. If equal, CMP merely returns. If not, the line, char position and byte number are reported. CMP can be run on any file.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO None.
NAME cp - copy files

SYNOPSIS cp file1 file2 cp file [files] directory/

DESCRIPTION COPY routine allows users to copy files from one place to another. It requires two or more arguments. The last argument is the destination file (or directory) and all others are source files. Like the 'mv' command, it accepts source arguments with wild cards and copies all matching files into the destination argument which should be a directory. For example:

    cp abc* xyz/

will copy all files whose names begin with abc into the directory named xyz. This is quite useful for backing up large groups of files. When copied, the name of the files will be unchanged in the new destination directory. Copy will overwrite any existing file by the same name unless it is r/w protected, a directory or a physical device driver.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO mv.
NAME demount - demount a disk volume

SYNOPSIS demount volume name

DESCRIPTION DEMOUNT allows the user to remove a previously mounted removable volume from the filesystem. It requires two arguments. The first is the numeric id of the volume and the second is the directory name by which it is known. The removable volume will become linked to the root directory and the directory name will be removed (provided that it can be found). For example,

    demount 1 user

will unlink the disk which has been mounted on drive 1 and remove the directory named "user".

WARNINGS The system must be reset if this command is executed from the volume being demounted.

BUGS None known.

FILES None.

SEE ALSO mount
NAME rmdir - remove directory lists

SYNOPSIS rmdir dir [dirs] ....

DESCRIPTION RMDIR permits directories to be removed. It accepts any number of arguments greater than or equal to one. Arguments may be pathnames and may include wild cards. If the argument is found, and is a directory containing only "." and "..", then it will be removed and will have its sectors deallocated. If the directory is not empty, it will not be removed.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO mkdir, mvdir, cd.
NAME du - summarize disk utilization

SYNOPSIS du [volume]

DESCRIPTION DU command summarizes disk usage. It counts the number of files on the disk and the number of sectors in use. The number of files includes directories, hidden files and all others. In other words, it really counts ALL files. The number of sectors in use includes those associated with the linked list used for file management. It can be called with up to one argument which is the volume number to summarize. If no argument is given, it will summarize the root volume. An example summary is given below.

130 files
  busy:  915, 229K, 79%
  idle:  240,  59K, 21%

For the busy and idle reports, the first value is disk sectors, the second value is in Kbytes stored on the disk, and the final value represents the percent busy or idle of the total volume capacity.

WARNINGS Unmounted volumes cannot be examined. Attempts to do so will require system reset.

BUGS None known.

FILES None.

SEE ALSO None.
NAME dup - duplicate a disk

SYNOPSIS dup

DESCRIPTION DUP allows the user to duplicate the disk currently mounted in the drive. It is intended for single disk systems only. Those with multiple drives can use the copy routine instead. It examines the disk i-map sectors and only loads those sectors which are actually in use. These sectors are buffered in memory until 32K of store is used. It then prompts the user to mount the slave disk, copies the store onto the disk and then prompts the user to mount the master disk. Using this scheme, at most 10 exchanges must be made. When completed, the slave disk will exactly match the master except for those sectors which were not in use. The content of these should not matter.

WARNINGS If any disk errors occur while writing the slave, dup cannot continue. Reset the system and repeat the operation. Be careful not to exchange the disks incorrectly.

BUGS None known.

FILES None.

SEE ALSO None.
NAME echo - type arguments as strings

SYNOPSIS echo [text]

DESCRIPTION ECHO simply sends all its arguments to the standard output as character strings. No character is placed between the strings when sent. Character translations such as \n are performed by the system when strings in double quotes are passed as arguments. If no arguments are given, ECHO simply types (exactly) the standard input to the standard output. Some examples follow.

    echo "Hello world"
    echo abc def "Now is the time"

ECHO is most useful when its output is redirected.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO None.
NAME fmt - format a disk volume

SYNOPSIS fmt [volume] [drive]

DESCRIPTION FMT allows users to format UNIX disks. If no argument is
given or if the argument is 0, FMT will create a root disk. If a
single argument is passed, it denotes the volume number by which the
newly created disk will be referred. If a second argument is given, it
is the drive on which the volume will be mounted for formatting. It
will request the insertion of the disk on the drive requested by the
user. After the new disk is inserted and a return is typed, FMT will
initialize the disk and place a fresh linked list having the directory
"." as well as the "con" driver in the directory if the disk is a root
disk. If the disk is not a root volume, it will contain "." and ".."
where ".." is linked to the root. After formatting, the user will be
asked to remount the system disk if drive 0 was used. Note that volume
and drive are independent. This allows the user to create any volume
using any drive.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO None.
NAME  inode - display inodes on a disk

SYNOPSIS  inode [number]

DESCRIPTION  INODE ROUTINE allows the user to read and modify disk inodes. To read an inode, simply type the inode number to display in decimal (10) or hex (0x10). To modify an inode, simply type the inode number, the element to modify in brackets, an equals sign, and the value to substitute. For example,

\[21(15)=0x44\]

Any value can be specified in decimal or hexadecimal. Inode accepts at most one argument which is used as the first one to dump. It will then prompt for subsequent inode values. To exit the routine, type end-of-file (NULL).

WARNINGS  None.

BUGS None known.

FILES None.

SEE ALSO None.
NAME ld - link edit load modules

SYNOPSIS ld ld_module [ld_modules] ....

DESCRIPTION LD is the system link editor. It accepts any number of arguments which are assumed to be names of ".o" files. The first of these is made the entry point of the resulting module so order counts. It automatically includes the file /lib/sys.o which links the operating system entry points (see LIB). For example,

   ld abc xyz /def/temp

will link edit abc.o, xyz.o, /def/temp.o, and /lib/sys.o into a single executable file named abc. If no errors occur, the result can be executed by simply typing its name. The resulting module is relocatable.

WARNINGS This routine is simply a shell program which calls the real link editor, xld.

BUGS None known.

FILES /lib/sys.o

SEE ALSO mk, cc, as, xld, reloc, LIB.
NAME ls - list files in directories

SYNOPSIS ls [-alstm] [file_names] [directory_paths] ....

DESCRIPTION The LS command allows the user to examine the contents of directories. It accepts as arguments either filenames to match or paths to directories. Any number of arguments of either type can be given. The command recognizes the wild card character (*) within filenames. Some examples follow.

```
ls
ls *.c
ls temp temp*
ls /bin/a*.c user/help
```

If a name ends in forward slash (/), ls will treat it as a directory and will list all files within it. Several flags are available to influence the command. These are described next.

```
-a List ALL files, even those marked as unlisted
-i List the starting inode and the name
-l Use the long listing. Gives the filemode byte, the starting inode (in decimal), the length (in hex) and the file name
-m List the filemode byte and name
-s List the file size (in hex) and name.
-t Trace and print all inode numbers (in decimal) used in the file(s).
```

Several flags can be given at once but they must be in alphabetical order. For instance,

```
ls -al
ls -t *.c
ls -at /user/help *.c ../bozo
```

Whenever the filemode byte is displayed, it is printed as the following symbols.

```
1 - No list option (override with ls -a)
* - Shell executable
```
p - Physical device driver
r - Read protected
w - Write protected
d - Directory

If the option is not enabled, an underline (_) is printed in the letter’s place.

WARNINGS None.

BUGS None known.

FILES The names . and .. indicate the current and previous directory respectively.

SEE ALSO chmod.
NAME  mk - make a C program

SYNOPSIS mk c_pgm [ld_files]

DESCRIPTION MK accepts a variable number of arguments. The first is expected to be the name of a C program, c_pgm.c, and any other arguments are expected to be load modules, ld_file.o. MK will run the C compiler and the assembler on the first argument. It will then run the link editor on all arguments. It automatically includes /lib/sys.o which is the system library. For example,

        mk xyz /lib/split ../findex

will compile xyz.c (which generates xyz.s), and assemble xyz.s (which generates xyz.o). Finally, it will load the files xyz.o /lib/split.o ../findex.o /lib/sys.o.

WARNINGS Remember to omit the ".c" and ".o" suffixes.

BUGS None known.

FILES None.

SEE ALSO cc,as,ld,xc,xc,xas,xld.
MKDIR

NAME mkdir - make new directories

SYNOPSIS mkdir name [name] ..... 

DESCRIPTION MKDIR routine permits the user to create new directories anywhere in the directory tree. Each argument should consist of a new directory name to place the in the parent directory. If the name is already taken or cannot be created for some reason, a message will be printed. Otherwise, the directory will be added and will contain links to itself and its parent directory. These two links will be marked as directorys themselves and will have the 'list' option off. The new directory name in the parent directory will be marked as a directory but will have the 'list' option on. Any number of arguments may be given.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO rmdir, cd
NAME mount - mount a removable volume

SYNOPSIS mount volume name

DESCRIPTION MOUNT allows the user to access a demountable volume. It requires two arguments. The first is the volume id to access and the second is the directory name which it is to be called. MOUNT then links the two disks so that the specified volume id appears as a subdirectory by the given name. MOUNT can be executed at any location in the filesystem. This allows a demountable volume to become a leaf in the root directory tree. If the given name already exists, the command will not mount the volume.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO demount
NAME mv - move files within the filesystem

SYNOPSIS mv file1 file2 mv file1 [files] directory/

DESCRIPTION MV command allows files (not directories) to be renamed or moved about in the directory hierarchy. The command requires at least two arguments. If more than two arguments are given, the last should be a path name to a directory. For example,

mv abc xyz def ..//dirx/ mv abc ..//dirx/

Wildcard (*) is also permitted. For example,

mv abc* xyz temp ../

The more simple (and usual) use is to rename a file. For example,

mv abc xyz mv temp oldfile

WARNINGS If several files are moved and the last argument is NOT a directory, all files but the last of them will be lost and the last will assume the new name. For example,

mv abc abcde abcdefg xyzzy mv abc* xyzzy

both result in lost files. A single file will remain named xyzzy which is the last one found when searching the directory. This command cannot be used to rename or relocate directories, files which are physical device drivers or protected files. To move such files, use chmod to clear the protection bits.

BUGS None known.

FILES None.

SEE ALSO cp, rm
NAME mvdir - move directories within the filesystem

SYNOPSIS mvdir old_dir new_dir mvdir dir [dir_list] directory/

DESCRIPTION MVDIR command allows directories (not files) to be renamed or moved about in the directory hierarchy. The command requires at least two arguments. If more than two arguments are given, the last should be a path name to a directory. For example,

mvdir abc xyz def ../dirx/ mvdir abc ../dirx/

Wildcard (*) is also permitted. For example,

mvdir abc* xyz temp ../

The more simple (and usual) use is to rename a directory. For example,

mvdir abc xyz mvdir temp oldfile

WARNINGS Never move a directory to itself (mvdir abc abc/), or to a subdirectory beneath it (mvdir abc xyz/), or operate on the names "." or "..". Either of these can scramble a disk beyond any hope of recovery.

BUGS None known.

FILES None.

SEE ALSO mkdir, rmdir, cd.
PR

NAME pr - print a list of files

SYNOPSIS pr [-h] [filenames]

DESCRIPTION PR does printing and reformatting of files. It accepts any number of arguments including none and treats each as the pathname to a file. Wild cards in the pathname are permitted. If used, PR will print all files which are found that match the pathname. It accepts a single flag, -h, which will suppress printing of the header on each page. If no matching files are found, an appropriate message is printed. Otherwise, the files will be printed at the standard output. If no arguments are given, PR will read the standard input for the information to print until it is exhausted. Tabs will be expanded as specified by the stty setting. The output of PR should be re-directed to the printer driver routine as shown below.

pr manual >/lpr

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO stty.
NAME pwd - print working directory

SYNOPSIS pwd

DESCRIPTION PWD prints the working directory. This is done by moving up the file system from the current directory and seeking directory names that have the inode of the current directory. The names are concatenated and printed when no "." directory is found. This denotes the top of the tree.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO cd, mkdir, rmdir, mvdir.
NAME reloc - relocate modules to an absolute address.

SYNOPSIS reloc file address

DESCRIPTION RELOC accepts two arguments. The first is the name of a module which has been link edited. The second is the address at which the module will be placed when it runs. The address can be specified in decimal (4096) or hexadecimal (0x1000). Upon completion, a file named core will exist in the current directory which contains the absolute image. For example,

reloc bozo 0xc000

will generate a core file which is bozo loaded at 0xc000. This command is most useful for generating ROM images.

WARNINGS Files generated for absolute loads will NOT execute under the operating system. The operating system memory management requires relocatable images which it converts to absolute images when the file is executed.

BUGS None known.

FILES core.

SEE ALSO None.
NAME  rm - remove lists of files

SYNOPSIS  rm file [files] ..... 

DESCRIPTION  RM permits files to be removed. It accepts any number of arguments greater than or equal to one. They may be pathnames and may include wildcards. The pathnames are split into directory and file. If the directory is found, any name in it matching the file will be removed and have its inodes marked idle. If no matching file is found in the directory, the filename is printed along with a message. If the directory is not found, the directory name is printed. Files which are read protected, write protected, physical device drivers, or directories cannot be removed. To remove such files, force their mode bits to 0 using chmod. Some examples follow.

rm  abc  rm abc* /bin/temp rm ../*yzy*.
c

WARNINGS  No second chance is given. Be careful with wild cards.

BUGS  None known.

FILES  None.

SEE ALSO  chmod.
RMDIR

NAME rmdir - remove directory lists

SYNOPSIS rmdir dir [dirs] ....

DESCRIPTION RMDIR permits directories to be removed. It accepts any number of arguments greater than or equal to one. Arguments may be pathnames and may include wild cards. If the argument is found, and is a directory containing only "." and "..", then it will be removed and will have its sectors deallocated. If the directory is not empty, it will not be removed.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO mkdir, mvdir, cd.
NAME sh - shell program

SYNOPSIS sh [files]

DESCRIPTION SH provides the basic system operation. It reads strings from the standard input which it passes to the SYSTEM routine for execution. It is the first procedure invoked by uNIX. It can be called with no arguments in which case it will read the standard input. If arguments are given, it will open each of them as standard input and execute them as SYSTEM commands until the file is exhausted. SH accepts no flags. SH is responsible for prompting for new commands. The prompt string can be changed using the stty command.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO »sh, system, stty.
STTY

NAME stty - set teletype options

SYNOPSIS stty [option [value] ]

DESCRIPTION STTY allows the user to setup various system options. The
current options are:

stty erase value (system character delete) stty kill value (system
line delete) stty tabs value (system tab stops) stty path string
(system command search) stty prompt string (system prompt string)

If no arguments are passed, STTY will print the value of all options.
If a single argument is passed, it will print the value of the
selected option. When the system is reset, the initial values of the
options are:

erase = 0x08 kill = 0x7f tabs = 0x04 path = "/bin/::" prompt = "% "

If tabs are set to 0, the system will print the true value of the tab
(0x09). If it is other than 0, the system will replace the tab with an
appropriate number of blanks for that group of tabstros. For further
discussion of the path variable, see the "system" subroutine
description.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO system.
NAME uucp - unix to unix copy

SYNOPSIS uucp [files] ....

DESCRIPTION UUCP permits transfer of blocks of files from one disk to another using memory as a buffer. It accepts any number of arguments (including none) and treats each as the pathname to a file. Wild cards in the pathname are permitted. If used, UUCP will load and buffer all files found that match the pathname. It requires no flags and if no matching files are found, an appropriate message is printed. After all files are found and loaded, a message will be printed requesting the user to exchange disks. When this is done, the buffered files will be written onto the new disk under their old names but they will be placed at the root directory. The system will re-start using the new disk.

WARNINGS No check is made for overflowing memory. Do not try to uucp more than 32K of files at a time.

BUGS None known.

FILES None.

SEE ALSO dup
NAME rmdir - remove directory lists

SYNOPSIS rmdir dir [dirs] ....

DESCRIPTION RMDIR permits directories to be removed. It accepts any number of arguments greater than or equal to one. Arguments may be pathnames and may include wild cards. If the argument is found, and is a directory containing only "." and "..", then it will be removed and will have its sectors deallocated. If the directory is not empty, it will not be removed.

WARNINGS None.

BUGS None known.

FILES None.

SEE ALSO mkdir, mvdir, cd.
BOOTSTRAP PROGRAM
FOR Z-80 MICROPROCESSOR
by
L. A. TOMKO

equates:

a_ equ 061h
b_ equ 062h
c_ equ 063h
d_ equ 064h
e_ equ 065h
f_ equ 066h
g_ equ 067h
h_ equ 068h
i_ equ 069h
j_ equ 06ah
k_ equ 06bh
l_ equ 06ch
m_ equ 06dh
n_ equ 06eh
o_ equ 06fh
p_ equ 070h
q_ equ 071h
r_ equ 072h
s_ equ 073h
t_ equ 074h
u_ equ 075h
v_ equ 076h
w_ equ 077h
x_ equ 078h
y_ equ 079h
z_ equ 07ah
sp_ equ 020h
nl_ equ 00ah
A_ equ 041h
B_ equ 042h
C_ equ 043h
D_ equ 044h
E_ equ 045h
F_ equ 046h
G_ equ 047h
H_ equ 048h
I_ equ 049h
J_ equ 04ah
K_ equ 04bh
L_ equ 04ch
M_ equ 04dh
N_ equ 04eh
O_ equ 04fh
P_ equ 050h
Q_ equ 051h
R_ equ 052h
S_ equ 053h
T_ equ 054h
U_ equ 055h
V_ equ 056h
W_ equ 057h
X_ equ 058h
Y_ equ 059h

e ‘a’
b ‘b’
c ‘c’
d ‘d’
e ‘e’
f ‘f’
g ‘g’
h ‘h’
i ‘i’
j ‘j’
k ‘k’
l ‘l’
m ‘m’
n ‘n’
o ‘o’
p ‘p’
q ‘q’
r ‘r’
s ‘s’
t ‘t’
u ‘u’
v ‘v’
w ‘w’
x ‘x’
y ‘y’
z ‘z’

All these were unnecessary —
I didn’t have documentation on the
cross-assembler, so I didn’t know
how to specify all characters.
operation code

org 0000h

starts at the reset location

0

make sure executing from ROM (cold start)

interrupt mode for Z-80 peripherals

number of 2-byte words in the CRT RAM

last location of the CRT RAM

two ASCII 'blank's, one in each byte

writes two 'blanks' to consecutive CRT locations

looking for h1=0 to stop clearing CRT RAM

test upper byte first

to make the flags appear

certainly not through if upper byte is > 0

now look at the lower byte

if both are zero, we are done with clearing CRT

Reset the disk controller, to abort SEEK command

which is ineffective because the head is

unloaded during the automatic Reset SEEK.
Now for a little egocentricity, we'll print a logo

```
ld h1,crtmax
ld de,0121h
add h1,de
ex de,h1
ld bc,lgosze
ld h1,log
ld dir
```

Enable the pio chip and the interrupts

```
ld a, pmod1
out (keyprt),a ; pmod1 = 4f, sets the pio to mode 1 (input)
ld a,kbvctr
out (keyprt),a ; keyport is the control port for keyboard pio
ld a,enable
out (keyprt),a ; enable = 083h, enables port interrupt
in a,(keybrd); do one read to set 'ready' output.
```

initialize 1ms timer. (used for cursor blinker)

```
ld a,07h
out (0f4h),a ; sets timer mode, non-interrupting, prescale/16
ld a,0fah
out (0f4h),a ; time constant of 250 counts = 1 ms total
ld a,08h
out (0f4h),a ; ctc vector(s) B + counter #
```

out (0cfh),a ; Set single density for floppy controller
xor a
ld (dskrdy),a ; a = 0
clear 'disk ready' flag

Initialize P10B, Port B for miscellaneous interrupts, including disk

```
ld a,0cfh
out (mscnt1),a ; Mode 3 - bit I/O
ld a,1
out (mscnt1),a
ld a,037h
out (mscnt1),a
ld a,0f9h
out (mscnt1),a
ld a,010h
out (mscnt1),a
in a,(mscnt)
```

do a read to set 'ready' output.

misc. initialization

```
xor a
ld (keyfg),a ; initialize keyboard flag = off
ld (kbwptr),a ; keyboard write pointer
ld (kbfptr),a ; keyboard read pointer
ld (cstat),a ; start with cursor status = off.
ld (dskid),a ; default disk ID is 00
ld (crtmod),a ; set CRT mode to normal
```
ld hl, crtram
ld bc, 0020h
add hl, bc
ld (curse), h1
call curson
ld hl, vectab
ld a, h
ld 1, a

: beginning of CRT RAM
: offset for the initial cursor position

ld h1, 0
ld de, 0
ld bc, 04000h
ldir
out (RAM), a

: start the cursor blinking
: the vector transfer table address is vectab (I hope)
: need upper byte in a to load reg. 1.
: load the high byte of vector table

ld sp, sp-base
ei
call restor

: Load RAM from ROM and change to RAM
: turns RAM on
: load stack pointer so we can call subroutines
: enable interrupts so disk will work
: set disk to track zero

now we ask the almighty z to monitor the keyboard.

monitor:
ld sp, sp-base
call dskdly
ld a, DSKRST
out (dskcmd), a
ld a, 0dh
call putcnt
ld a, 024h
call putcnt

: reload the stack pointer
: wait for controller to clear
: Reset controller, abort any commands
: first print a 'cr' and a 's' on the crt

: enable the system interrupts.
: This next little operation clears peripheral devices
: that may have pending interrupts acknowledged but
: not cleared with a "reti" command. Each iteration
: clears only one device, so we will do three just
: to be sure!

el!
ld hl, mon1
push hl
reti
mon1:
ld hl, mon2
push hl
reti
mon2:
ld hl, mon3
push hl
reti
mon3:

: call kbdchr
: and 07fh
: cp 072h
: jp z, readkw
: cp 077h
: jp z, writekw
: cp 065h
: jp z, execute
: cp b,
: jp z, uboot
: ld a, qp
: call putcnt
: jp monitor

: that's 'keyboard character' - which returns a
: character when a key is depressed,
: masks the parity bit (maybe not necessary)
: is it an 'r'?
: if so, it may be a read command
: maybe a WRITE command
: maybe a write command
: how about an 'e'?
: probably an execute command
: 'b' points to boot command
: if anything else, we load a '?' into A.
: and dump it to the crt at the cursor position
: then try again

we seem to have a 'read' command, but let's
: look at the second character to find out which kind

readkw: call kbdchr

Here's the RAM/NROM transfer, writes to low order (0-4000) are
always allowed. Reads start next instruction after turn-on.
and 07fh
cp 06bh
jp z, rdkbd
jp 074h
jp z, rdtape
cp 064h
jp z, rddsk
cp 06dh
jp z, rdmem
ld a, 03fh
call putcr
cmpion
:
:
rdmem: call getaddr
push hl
:
:
call getaddr
push hl
:
pop bc
:
pop hl
:
ld a, (hl)
:
call phxch
ld a, sp_
call putcr
inc hl
:
dec bc
:
ld a, b
:
or c
:
jp z, monitor
:
jp rdmi

rdmi is a routine to allow input of hex data into consecutive memory locations beginning with the address specified after the keyword 'rk', followed by a space. The hex bytes are separated by blanks, may appear as many per line as desired (and fit). A prompt consisting of the next address appears after each newline. The sequence is terminated by an 'eof' character.

rdkbd: call getaddr
:
rkbrtr:
call kbdchr
:
and 07fh
cp 02eh
:
jp z, n byte
:
jp 06h
:
cp 0dh
:
jp z, prompt
:
jp 04h
:
jp z, monitor
:
hrex
:
call putcr
:
jp rkbrtr

nbyte looks for two hex characters. If successful, it will write the equivalent byte in the current location in 'addr'. If garbage is received, '?', will be displayed until a 'blank', newline or eof - i.e., the particular byte is aborted.

n byte: call kbd chr
:
call chk hex
:
cp 0ffh
:
jp z, abtr
:
print a '?'. and look for a blank, eof or newline
ld b,a
    ; tuck the hex digit away for a microsecond
    call kbdchr
    ; get the next digit
    call c1khex
    ; same song second verse
    cp 0ffh
    ;
    jp z,abrt
    ;
    ld hl,(addr)
    ;
    rrd
    ; that's where we want to store this byte
    ld a,b
    ; recall, that's the upper nibble
    rrd
    ; prastol the lower nibble shifts to its proper place,
    ; and the upper nibble slips in behind it. deos ex machine!
    inc hl
    ; get the next address location
    ld (addr),hl
    ; and store that in addr
    jp rkbstrt
    ; look for blanks, etc.

    writkw:
    ; Write command - tape or disk? or mistake?
    call kbdchr
    ; Get the next character from keyboard
    and 07fh
    ; mask parity bit
    cp 074h
    ; 't' for tape
    jp z,wrtape
    ;
    cp 064h
    ; 'd' for disk
    jp z,wdsn
    ;
    cp 08dh
    ; 'm' for memory (block move)
    jp z,blkmv
    ;
    ld a,03fh
    ; '?' if we don't know what else to do
    call putcrt
    ;
    jp monito
    ; abort if unrecognizable character sequence

    Block Move routine moves from source address to destination
    address as many bytes as you please.

    blkmv:
    ; next word will be the "from" address
    call getaddr
    ; save it
    push hl
    ; next word will be "to" address
    call getaddr
    ; finally, how many bytes to transfer?
    push hl
    ; BC is the counter for the LDIR command
    pop bc
    ; DE is the destination pointer
    pop de
    ; and HL is the source.
    pop hl
    ; GO!
    ldir
    ; done, go home.

    wrtpe:
    ; Write memory to tape. First find out from where
    ; to read, where to stop, and where to write.
    call gtprms
    ; Big tape writing routine. May be a dummy at first.
    call wtap
    ; Back to monitor when finished
    ;
    wdsk:
    ; Write block to disk. May be a dummy at first.
    ;
    ld c,1
    ; 'Write' flag to common disk I/O routine
    call dskto
    ;
    ;
    rddsk:
    ; Read block from disk
    ; Set 'Read' flag to common disk I/O routine
    ;
    ;
    execute:
    ; Jumps to address specified after 'ex' keyword
    call kbdchr
    ; Get character after the 'e'
    cp 078h
    ; Should be an 'x'
```assembly
jp nz,abrt
ld bc,monitor
push bc
call getaddr
jp (hl)

; Want returns to go to monitor
; so load that to the stack.

; Gets the next 4 hex characters, assembles them
; into address in HL, to which we journey now. bye!

uboot:
call kbdchr
cp t
jp z, uboot1
ld a, qm
call pulcrt
jp monitor

uboot1:
ld bc, 0f00h
ld de, 0a100h
ld hl, 01000h
ld dir

ld h, 0c000h
ld (strtlc), hl
ld h, 040h
ld (enloc), hl
ld h, 3
ld (tnode), hl
ld a, 0
ld (rbufg), a
call dsxio!

ld bc, 0fh
ld de, 0c057h
ld hl, 0a920h
ld dir

jp 0a100h

end monitor

-------------------------------------
timer:
push af
push bc
ld b, a
ld a, 0c7h
out (t2), a
xor a
ld (timflag), a
ld a, b
out (t2), a

: CTC 2 is driven by CTC 0
: Tuck away the input parameter
: setup CTC 2 as counter, with interrupt enabled
: t2 = 0f6h = timer 2
: A = 0
: zero timer flag
: recall input parameter
: The count (1-255)
: now loop till timer flag is turned on again
: Clean up and return when flag is set (by the
: 'timeln' interrupt handler routine

jnz timlp:
push af
ld a, 04h
out (t2), a

: Here it is: handles the CTC-2 interrupts for timer
: Disables counter interrupt and resets it
```
ld a,1
ld (timflag),a
pop af
ei
ret

end timein

; Times for 'a' x 250 milliseconds
push bc
ld b,a
ld a,250
ltp:
call timer
dec b
jp nz,ltp
pop bc
ret

end ltp

; Returns 2 ASCII characters in h and l, representing the hex byte in 'a'
push af
push bc
ld b,a
and 0fh
cp 10
jp m,digit
sub 10
add a,061h
id l,a
jp thigh
digit:
add a,030h
ld l,a
thigh:
ld a,b
rrc a
rrc a
rrc a
and 0fh
cp 10
jp m,dig2
sub 10
add a,061h
ld h,a
jp thru
dig2:
add a,030h
ld h,a
thru:
pop bc
pop af
ret

end hexchr

; kbdchr returns a character when and if a key is depressed.
; save environment, stash all garbage!
push bc
push hl

keyloop:
  ld a,(keyflg)
  or a
  jp z,keyloop
  ld a,(kbdptr)
  inc a
  and kbfsze
  ld (kbdptr),a
  ld c,a
  ld b,0h
  ld h1,kbbuf
  add h1,bc
  ld a,(kbwptr)
  cp c
  xor nz,rdout
  ld (keyflg),a
  ret

rdout:
  ld a,(hl)
  pop hl
  pop bc
  ret

End kbdchr

******************************************************************************

getaddr is a routine to get 4 hex characters and assemble them into an address in HL and 'addr'. Aborts to monitor if it doesn't understand anything.

getaddr:
push af
  call kbdchr
  and 07fh
  cp 020h
  jp nz,qmk
  call kbdchr
  call chkhex
  cp 0ffh
  jq z,qmk
  ld h1,addr
  inc h1
  ret

  call kbdchr
  call chkhex
  cp 0ffh
  jq z,qmk
  ld h1,addr
  inc h1
  rld

  call kbdchr
  call chkhex
  cp 0ffh
  jq z,qmk
  ld h1,addr
  rld

  call kbdchr
  call chkhex
  cp 0ffh
  jq z,qmk
  ld h1,addr
  rld

  call kbdchr
  call chkhex
  cp 0ffh
  jq z,qmk
  ld h1,addr
  rld

  call kbdchr
  call chkhex
  cp 0ffh
  jq z,qmk
  ld h1,addr
  rld

  call kbdchr
  call chkhex
  cp 0ffh
  jq z,qmk
  ld h1,addr
  rld

  call kbdchr
  call chkhex
  cp 0ffh
  jq z,qmk
  ld h1,addr
  rld
`.jp z, qmk
ld hl, addr
rld

ld hl, (addr) ; return number in hl as well as (addr)
pop af
ret

qmk:
ld a, 03f
 call putcrt ; write it to crt.
jp monitor ; abort and return to the monitor.

End getaddr

prompt:
ld a, 0dh
 call putcrt
ld de, (addr)
ld a, d
 call pthxch
ld a, e
 call pthxch
ld a, 03ah
 call putcrt
ld a, 020h
 call putcrt
jp nxbyte

This is part of monitor: prompt

getprms:
call getaddr
ld (strtic), hl
 call getaddr
ld (endloc), hl
 call getaddr
ld (tnode), hl
ret

asci:
push de
ex de, hl
ld hl, temp1
ld a, d
 call chkhx
 cp Offh
jp z, hexerr
rld
ld a, e
 call chkhx
 cp Offh
jp z, hexerr
rld
ld a, (temp1)
jp dnasc

hexerr:
ld hl, msg3
 call putstring
ld a, d
 call putcrt

msg3 = 'Invalid hex char'
puts the string to CRT
NOW PRINT THE input data
ld a,e
  call putcrt
  xor a
  ret

; return with 0 in A

donasc: pop de
  ret

;------------------------------------------
; msg3:
  db 049h,06eh,076h,061h,06ch,069h,064h,020h,068h,065h
  db 078h,020h,063h,068h,061h,072h,020h,0

; End ASCII
;------------------------------------------

chkhex:
  cp 030h
  jr m,erhx
  cp 03ah
  jr m,numhx
  cp 061h
  jr m,erhx
  cp 067h
  jr m,alphx

erhx:
  ld a,0ffh
  ff -> A
  ret

numhx:
  sub 030h
  A - '0' --> A
  ret

alphx:
  sub 057h
  (A - 'a') + 10 --> A
  ret

; End chkhex

;------------------------------------------

putcrt writes a character (found in a) to the CRT
much like a serial terminal.

putcrt: push af
  push bc
  push de
  push hl
  call cursoff
  cp ESC
  jr z,se		 ; First look for the ESC character
  id c,a
  id a,(crtmod)
  and ?
  cp ESCMOD
  jr z,escape
  cp ROWMOD
  jr z,row
  cp COLMOD
  jr z,column
  id a,(crtmod)
  cp INSMOD
  jr z,insert
  or a
  ; Turn the cursor off, which replaces the character
  ; at the cursor location and prevents interrupts.
  ; If found, set escape mode
  ; save the character in C for a while
  ; Find state of CRT
  ; look at all but INSERT mode bit
  ; escape mode?
  ; Row mode?
  ; column mode?
  ; reload CRT mode to look at INSERT bit
  ; insert mode.
call nz,audcrt1
1d a,c
: Any other modes are illegal
: restore character to A
cp 020h
: check for special chars (< 20h)
jp m,special
: handle those separately
cp 080h
: currently not allowing bit 7 = 1
jp p,special :
jp z,special :
1d hl,(curse) :
: pointer to cursor
1d (hl),a :
: that's where we will write.
now, fiddle with the cursor.
1d a,l :
: lower byte of cursor address
cp 04fh:
: that's eos for even rows
jp z,eoin :
: return and scroll
cp 0c fh :
: eos for odd lines
jp z,eoin :
inc a :
: if not eos, just increment cursor address
1d l,a :
: don't worry about carry - never occurs in line.
1d (curse),hl :
: update the cursor position
nullo: call curson :
pop hl :
pop de :
pop bc :
pop af :
ret :
: bye!

eoin: 1d de,031h :
: just wrote last char on line, move cursor to
add hl,de :
beginning of next line. (by adding 31h)
1d (curse),hl :
: update the cursor position
1d de,botin :
: if cursor >= xDxx, we must scroll
1d a,d :
: just look at the upper byte
cp h :
jp nz,nullo :
: otherwise, we'll just return
jp nz,botift :
: scroll needed - first set cursor to bottom left
1d (curse),hl :
: scroll moves everything up one, but leaves cursor.
call scroll :
jp nullo :
special:
: handles special characters, like tabs, spaces, etc.
cp 080h :
jp p,nullo :
jp z,nullo :
: initially, if bit7 = 1 we will just ignore it.
cp 08h :
: backspace?
jp z,backsp :
: cp 09h :
: tab?
jp z,tab :
cp 0ah :
: linefeed?
jp z,lf :
cp 0ch :
: formfeed (clear screen)?
jp z,clr :
cp 0dh :
: carriage return?
jp z,cr :
jp nullo :
: don't know what it is, so just ignore it.

backsp: 1d hl,(curse) :
: backspace moves cursor back, does not erase.
1d a,l :
: stops backspacing at beginning of line.
cp 0h :
jp z,nullo :
cp 040h :
dec a
ld 1,a
ld (curse),hl
jp nullo

: put out spaces until cursor is left at even '8' mult.
ld a,020h
call putcrt
ld hl,(curse)
ld a,1
and 07h
jp z,nullo
jp tab
: look at last 3 bits
done.
do it again

if:
: leaves cursor in same relative position.
ld de,080h
add hl,de
ld a,h
however, if cursor >= 0300h, must scroll
: compare bottom of screen (high byte)
cp hbot
: if not below screen, reload cursor
call scroll
: scroll won't change curse, so we just leave it alone
jp nz,nxtline
: gracefully return.
pnullo
nxtline:
: reload the cursor with the new value
ld (curse),hl
jp nullo

clr:
: going to do 26 scrolls to clear the screen
ld a,26
call scroll
: going to put cursor at top left
dec a
jp nz,movup
ld hl,uplft
ld (curse),hl
jp nullo
: graceful exit.

cr:
: return to start of line and do line feed
ld a,1
look at the lower byte
cp 07fh
: if greater than 7fh, we were on an odd line
jp p,oddin
: even line: return to 0,(l)
ld 1,0h
ld (curse),hl
: moves the cursor back to start of current line
jp 1f
oddim:
: now do a line feed. note: routine returns 'cr'
ld 1,080h
ld (curse),hl
jp 1f

: escape:
: previous character was and ESC
ld a,(crtmod)
and INSMOD
ld (crtmod),a
ld a,c
: clear all but INSERT mode bit
cp E,
jp z,clr
: look at the new character
cp K,
jp z,clr
: Home/Clear routine
ineline
: Clear to end of line
jp z,insline
: insert new line at cursor row
cp Mon2
jp z,dele
  cp amp_
  jp z,insmod
  cp 0
  jp z,exmod
  cp Nh
  jp z,delchr
  cp Y_
  jp z, murow
  jp nullo
: delete a line
: enter insert mode
: (capital 0)
: exit insert mode.
: Delete character at cursor
: cursor row set
: ignore anything we don't understand

clr:  

1d h1,(curse)  : Clears from cursor to end of line

cl1:
  1d a,1
  cp 04fh
  jp z,nullo
  cp 0cfh
  jp z,nullo
  ld a,sp_
  not done, so write a blank
  ld (h1),a
  at (h1)
  inc 1
  Next...
  jp cl1
  do it again

ins:
  call crsbro
  : Inserts blank line at current cursor
  : line, moving lines below it down.
  : cursor to beginning of line, --> HL

ins0:
  1d de,bot1ft
  scf
  ccf
  sbc h1,de
  jp z,clr
  is cursor at bottom row?
  if so, just clear the line
  1d h1,bot1ft-080h
  : Next row up

ins1:
  1d bc,80
  ldir
  : 80 columns
  moves 'em down
  scf
  ccf
  ld bc,0d0h
  ex de,hi
  sbc h1,bc
  ex de,hl
  sbc h1,bc
  ld bc,(curse)
  ld a,e
  going to see if the activity is at (curse)
  cp c
  jp nz,ins1
  if not, do some more
  ld a,d
  cp b
  jp nz,ins1
  call crsbro
: blank the row if at cursor

dele:
: Deletes line of cursor; moves lines below
: one up; blank line at bottom; cursor
: at beginning of deleted line.
: cursor to beginning of current line
: HL now has starting cursor position

    call crsbro
    ld d,h
    ld e,1
ld bc, 080h
add hl, bc
ld bc, 80
call repeat
jp nullo

Next line below
80 columns
part of scroll: routine.

crsbol:

push af
ld hl, (curse)
lld a, 1
cp 07fh
jp p, crsb1
ld 1, 0
ld (curse), h
pop af
ret

crsb1:

ld 1, 080h
ld (curse), h
pop af
ret

BOL for odd line = xx80

BOL for even line

Insmod:

ld a, INSMOD
ld (crtmod), a
jp nullo

Enter the insert mode

Movrow:

ld a, (crtmod)
or ROWMOD
ld (crtmod), a
jp nullo

Enter row address mode

Put rowmode bits in without changing insert mode

Setesc:

ld a, (crtmod)
or ESCMOD
ld (crtmod), a
jp nullo

An ESC character has been received; set esc mode

Exmod:

xor a
ld (crtmod), a
jp nullo

Exit insert mode - clear everything

Delechr:

ld hl, (curse)

Delete character at cursor position.

Move text right of cursor left one slot.

delec1:

ld a, 1
cp 04fh
jp z, delec2
inc hl

End of line, even row

End of line, odd row
ld a,(h1)
inc h1
ld (h1),a
jp delc1

delc2:
ld a,sp
ld (h1),a
jp nullo

: Blank last character in row

row:
ld a,c
cp 25
jp m,row1
ld a,24
: load row register
: only 25 rows
: OK if less than 25
: else set to max = 24
row1:
ld (rowno),a
ld a,(crtmod)
and INSMOD
or COLOMD
ld (crtmod),a
jp nullo

: save it for next time
: clear all but insert mode bit
: concatenate column mode bits

column:
ld a,(crtmod)
and INSMOD
ld (crtmod),a
ld a,c
cp 80
jp m,col1
ld c,79
: expect column address
: clear all but insert mode
: only 80 columns
: if less, OK
: else change C to 79
col1:
ld de,080h
ld hl,up1ft
ld a,(rowno)
or a
jp z,col3
add hl,de
dec a
jp nz,col2
: distance between rows
: upper left of screen
: row number from previous read
: skip multiply if rowno = 0
: crude multiplication
: add A times!
col2:
ld a,c
add a,1
ld h,d
ld (curse),h
jp nullo
: column offset
: add it to the hl (don't worry about carry)

insert:
ld de,(curse)
ld h,d
ld a,e
cp 80
jp p,insrt1
ld 1,04fh
: inserts a character at cursor position
: won't advance beyond EOL
: > 80 --> odd line
: EOL for even line
insrt1:                ; EOL for odd line
    ld #1,0cfh
insrt2:                ; Are we back to cursor position yet?
    ld a,e
    cp 1
    jr z,insrt3
    dec 1
    ld a,(hl)
    inc 1
    ld (hl),a
    dec 1
    jr insrt2
insrt3:                ; load the incoming character there
    ld (hl),c
    ld a,#1
    cp 04fh
    jr z,insrt4
    cp 0cfh
    jr z,insrt4
    inc h1
    ld (curse),h1       ; load cursor
    jr null0

End of putcrt

*********************

aundcr1:              ; "crtmod" had illegal value
    push af
    push hl
    ld hl,crtm1
    call pltstrng
    ld a,(crtmod)
    ld 1,a
    xor a
    ld (crtmod),a
    ld a,1
    call putcrt
    pop hl
    pop af
    ret

crtm1:                 ;
    db 1,1,1,_e_,g_,a_,_l_,sp_,c_,R_,T_,sp_,m_,o_,d_,e_,nl_
    db 0

*****************************
curses:               ; cursor blinker, handles 250 ms interrupt.
    push af
    push bc
    push hl
    ld a,(cstat)
    call pltstrng
    jr z,turnoff
    ld a,(cstat)
    or a
    jr null0

    ld a,(curse)
    ld hl,(curse)
ld (hl),a ; and write the character there
xor a ; a = 0
ld (cstat),a ; clear cursor status.
jp home ; return
:
turnon:
ld h1,(curse) ; find the cursor
ld a,(h1) ; get the character there
ld bc,cchar ; character storage address
ld (bc),a ; store the character there
ld a,05fh ; an underscore (the cursor)
ld (h1),a ; put it on the crt
ld a,01h ; a = 1
ld (cstat),a ; set cursor status = 'on' (1)
:
home:
pop hl ;
pop bc ;
pop af ;
el ;
ret ;
:
End curses

******************************************************************************
cursoff:
push af ; turns cursor off by stopping counter, making sure
cursoff:
push af
push hl
ld a,043h
out (0f7h),a ; f7 is ctc port 3 (cursor counter)
ld a,(cstat) ; find out what state the cursor was in
or a
jp z,donoff ; cursor already off, so just quit
ld a,(cchar) ; cursor was on, so get character from storage
ld h1,(curse) ; find the cursor
ld (h1),a ; put 'er there!
xor a ; now clear the cursor status bit
:
donoff:
pop hl
pop af ;
ret ;
:
End cursoff

******************************************************************************
curson: push af ; turns the cursor on.
curson:
push af
ld a,(cstat)
or a
jp nz,donon ; if so, quit.
ld a,0c7h
out (0f7h),a ; if not, set ctc to go
ld a,tau
out (0f7h),a ; tau = time constant in ms (0fah = 250 ms)
:
donon:
pop af ; Just to make sure, enable interrupts
el
ret ;
:
End curson

******************************************************************************
scroll: push af ; scroll moves everything up one notch on crt.
scroll:
push bc
push de
push hl
call curson

ld bc, 050h
ld de, uplff
ld hl, scdline
call repeat
pop hl
pop de
pop bc
pop af
ret

repeat:
ld bc, 030h
add hl, bc
push hl
ex de, hl
add hl, bc
ex de, hl
pop hl
retreived.
ld bc, 050h
ld a, h
cp hibot
jp nz, repeat
ld hl, botlff
ld de, botlff
ld (hl), 020h
ld dir
pop hl
ret

End repeat

keyint:
push af
push bc
push hl
in a, (keybrd)
cp 04
jp 04, keysoft
cp 018h
jp z, keynt
inc a
inc kbdsize
ld (kbwptr), a
ld h1, kbdbuf
add a, 1
ld 1, a
ld (hl), c
ld a, 1
ld (keyflag), a
pop hl
pop bc
pop af
ret

keysoft:
ld hl, monitor
push hl
ret

keyinr:
ld hl,0
push hl
ret

End keyinr

pstrng:
push af
: Put ASCII string found in location HL to CRT until '0' byte encountered
strlp: ld a,(hl)
: look at each character
cp 0
: The null character
jp z,strgtr
: quit when zero encountered
call putcrt
inc hl
: Else write it to CRT
jp strglp

strgtr: pop af
: next one
ret

End pstrng

pthxch:
push hl
: Puts hex number to CRT as two ASCII characters
call hexchr
ld a,h
: returns two ASCII chars in H,L
ld a,l
: High one
call putcrt
ld a,l
: call putcrt
pop hl
: pop HL
ret

End pthxch

ldsv:
ex de,hl
pop hl
ex (sp),hl
push de
ld de,15
ex de,hl
ld de,15
call cdiv
inc a
ld d,1
ld c,h
pop hl
ld a,(dskid)
cp c
jp nz,errmsg4
ld a,d
out (dskdata),a
ld a,SEEK
out (dskcmd),a
call waitdk

: Preparation for disk read or write
: Save address in DE (stolen from SMW)
: That's the subroutine return address
: pops Tnode from stack and pushes return address
: Also save the from/to address on stack
: Number of sectors per track
: our divide routine divides de/hl
: From Z-80 library
: 1 + remainder = sector
: now track = d
: c = disk number (use 0 as default)
: now HL contains the address
: The current disk id, possibly loaded by hand
: If not right disk, abort w message
: The desired track
: Load to disk data register
: SEEK = 0fh, the seek command
: send it to controller
: Waits for disk controller to interrupt
in a,(dskcmd) ; get status
and 018h ; look at CRC or seek errors
jp nz,errmsg5
ret

errmsg5: ld h1,msg5
"Seek error"
call pstrng
erdate: ld a,d
callpthxch
ld a,e
sector
callpthxch
jpmonitor

msg5: db 053h, 065h, 065h, 066h, 020h, 065h, 072h, 072h, 06fh, 020h, 0

End lds
;**************
save:
; Writes one block to disk
; given address in HL, and Tnode second on
; stack.
; Return address
pop bc
pop de
Tnode
push bc
push de
Save the return address
callldsv
; Give next routine Tnode in stack
callkeyoff
id a,0b7h
; enable PIO int; 'or' logic; active high; mask follows
out(mscntl),a
ld a,0feh
out(mscntl),a
Mask all but D0.
ld a,014h
out(mscntl),a
Vector --> dskout interrupt handler.
in a,(mscio)
; clears PIO ready signal.
lde, E should contain sector number
out(dsksectl),a
ld b,0fh
ld c,0d3h
set C to controller data port
ld a,WRSCTR
write sector command (0a8h)
out(dskcmdl),a
to the command register
callwaitdk
now just sit back and wait for disk to finish
callcurson
; let 'er blink again
in a,(dskcmd)
and 07ch
mask the write-pertinent bits
jpnz,errmsg7
Print write-error message if any bad bits
ret
otherwise, return

errmsg7: ld h1,msg7
"disk write error"
callpstrng
ld ad, should contain Tnode
callpthxch
write it to CRT
ld a,e
lower byte
callpthxch
jpmonitor

msg7: db 044h, 069h, 073h, 06bh, 020h, 077h, 072h, 069h, 074h, 065h
    db 020h, 065h, 072h, 072h, 06fh, 072h, 020h, 0

End save
keyon:  push af
      ld a,enable
      out (keyprt),a
      pop af
      ret

keyoff: push af
      ld a,disable
      out (keyprt),a
      pop af
      ret

\-----------------------------

dskout:  push af
      push bc
      push de
      ld c,0d3h
      ld a,0b7h
      out (mscnt1),a
      out (mscnt1),ld b,0f9h
      out (mscnt1),ld a,0b8h
      call dsksc
      ld a,00fh
      out (mscnt1),a
      in a,(mscio)
      ld a,e
      out (dskcmd),a
      call wai
      call curson
      call keyon
      in a,(dskcmd)
      and 01ch
      call nz,erprint
      jp nz,errmsg6

; Disk output interrupt handler.
; NOTE: this has to be FAST!
; INPUTS: HL contains the address to be written
; from. C contains the disk controller data port.
; B should be written to a high value to prevent
; it from decrementing to zero, which would
; set the "z" flag and screw up the dkwait
; routine.
; outl (HL) > (C); inc HL; dec B
; in a,(mscnt1) clears the PIO interrupt.
; ret

end dskin

\-----------------------------

load:  pop bc
      pop de
      push bc
      push de
      call lds
      call cursof
      call keyoff
      ld c,0d3h
      ld a,0b7h
      out (mscnt1),a
      ld a,0f9h
      out (mscnt1),a
      ld a,012h
      out (mscnt1),a
      in a,(mscio)
      ld a,e
      out (dskcmd),a
      ld b,0ffh
      ld a,RDSTC
      out (dskcmd),a
      call wai
call curson
call keyon
in a,(dskcmd)
and 01ch
call nz,erprint
jp nz,errmsg6

; Reads a sector to memory specified in HL
; from disk at Tnode, contained in stack

; First exchange the first two stack positions

; Finds the right track and disk, and unravels Tnode
; We don't want interrupts during disk read.

; Don't want keyboard interrupts, either.
; Setup C for the "diskin" interrupt handler.
; Enable int, 'or' logic, active high, mask follows

; Mask all but DO.

; Vector to "diskin"

; Do a read to set ready flag.
; E should contain the sector number
; Load it in controller sector register
; B will decrement and affect flags at 0
; Command to read a sector (= 0b4h)
; To command port
; Wait for execution
; Restart cursor - interrupts OK now.

; Get status
; Mask CRC and lost data bits
; Print disk read-error message if bad bits
ret : else done - return

errmsg: ld hl, msg6
call print
ld a, d
ld pthxch
ld a, e
ld pthxch
jp monitor

msg6: db _, i, e, k, sp, r, e, a, d, sp, e, r, r, o, r, nl
       db t, r, e, c, k, sp, s, e, c, t, o, r, sp, 0

end load

************************************************************************
dskin:
        Interrupt handler for disk reads.
        NOTE: this routine must go FAST!
        INPUTS: HL contains the start address to be
                loaded to. It will be incremented each call.
                C contains the input port address.
                B will be decremented each time, so
                Flag register will be affected when
                B becomes 0. This may interact with calling
                program. Set B to 'FF' before starting to
                minimize this.

        ini
        : input (C) and Inc HL. (16 cycles)
        : (4 cycles) This is needed temporarily to
        : prevent bad memory read of 'ff'.
        ei
        : (4 cycles)
        reti
        : (14 cycles) resets PIO without reading it.

************************************************************************
dskio: ld a, c
       ld (rwfig), a
       call gtprms

dskio1: ld hl, (endloc)
       ld a, 1
       or a
       jp nz, wdsk1
       ld 1, 1
       ld (endloc), hl
       push hl
       ld hl, (tnode)
       ld a, (strtlc)
       or a
       jp z, rd
       call save
       jp mop

rd: call load
mop: ld hl, (endloc)
       dec 1
       ret z
       ld (endloc), hl
       ld de, 0100
       ld hl, (strtlc)
       add hl, de
       ld (strtlc), hl

ENTRY: dskio1 for uboot operation, else dskio

INPUT: C contains read/write flag (0=write).

Common routine to read/write 'endloc' blocks
from/to disk to/from memory beginning at
'strtlc'. The disk sectors are sequential,
beginning at remainder of tnode/15 + 1

on track tnode/15 low byte.

if endloc = 0 or 1, a single block is written

just set endloc = 1 if it was zero on input

'save' and 'load' expect tnode in stack

and starting address in HL.

C contains a write/read flag (0 = write).

Write a block to disk

Reads a block from disk

mopup: see if we are through

reduce number of blocks left to go by one

go home if done

resave endloc if not done.

add 256 to the start location
ld hl, tnode ; prepare to increment tnode (for next sequential block)
inc (hl)
jp wsk1 ; then do it again

End dskio

******************************************************************************

waitdk:
push af
ld a, 7

waitdk1:
dec a
jp nz, waitdk1

waitdk2:
ld b, 0ffh ; 8 is decremented (but not tested) in disk interrupt routines (diskin, diskout). We
write 8 to ff to prevent a 'z' condition in the flag register in case the interrupt occurs during the "and" operation. We may therefore miss the resetting of the 'busy' status bit for one loop, but no matter.
in a, (dskcmd) ; now start looking at controller status register
and 1
jp nz, waitdk2
pop af
ret

End waitdk

******************************************************************************

ccdiv:
ld b, h
ld c, l
ld a, d
xor b
push af
ld a, d
or a
call m, cccdeneg
ld e, b
or a
call m, ccbcneg
ld a, 16
push af
ex de, hl
ld de, 0
ccdiv1: add hl, hl
call ccrdel
jp z, ccdiv2
call ccmppcde
jp m, ccdiv2
ld a, 1
or 1
ld l, a
ld a, e
sub c
ld e, a
ld a, d
ld a, d
sbc a, b
ld d, a
ccdiv2: pop af
dec a
jp z, ccdiv3
push af
jp ccdiv1

ccdiv3: pop af
ret p
call ccdeneg
ex de,hl
call ccdeneg
ex de,hl
ret

ccdeneg: ld a,d
  cpl
  ld d,a
  ld a,e
  cpl
  ld e,a
  inc de
ret

ccbcneg: ld a,b
  cpl
  ld b,a
  ld a,c
  cpl
  ld c,a
  inc bc
ret

ccrdel: ld a,e
  rla
  ld e,a
  ld a,d
  rla
  ld d,a
  or e
ret

ccmpbdc: ld a,e
  sub c
  ld a,d
  sbc a,b
ret

wtape: ret ; dummy tape write program
rdtape: ret ; Dummy tape read program

logo: db 04ch, 041h, 054h, 027h, 073h, 020h
db 046h, 06fh, 06ch, 06ch, 079h, 0h

org 0900h ; origin of vector table for interrupts
vectab: dw keyinr
  00 > keyboard interrupt
  02 > tape output - not implemented yet
dw vcterr
  04 DMA controller - not supposed to interrupt
dw vcterr
  06 > tape input - not implemented yet.
dw vcterr
  08 > CTC channel 0 - shouldn't interrupt.
dw vcterr
  0a > CTC channel 1 - not defined yet.
dw timein
  0c > timer channel.
dw curses
  0e > cursor interrupt.
dw vcterr
  10 > PIO interrupt - formerly DISK controller.
dw dskin
  12 > PIO for disk input operations.
dw dskout
  14 > Same PIO for disk output operations
routine to handle strange vectors from interrupts.

:\push\ hl
 1d hl, msg8
 1call pstrlen
 1pop hl
 1ei
 1ret

:\msg8: db nl, u, n, k, n, o, w, n, sp, l, n, t, e, r, r, u, p, t, sp
      db v, e, c, t, o, r, r, nl, 0

:\end vectab

ERROR MESSAGES

:\errmsg: 1ld hl, errmsg
 1call pstrlen
 1call pthxch
 1ld a, c
 1call pthxch
 1jp monitor

:\msg4: db 057h, 061h, 06eh, 074h, 065h, 064h, 020h, 064h, 069h, 073h, 06bh
       db 02fh, 046h, 066h, 075h, 06eh, 020h, 064h, 069h, 073h, 06bh
       db 03ah, 020h, 0
 1format: A disk formatting program for 8"
       single density single side disk.

:\format: 1call restor
         1ld d, 0
 1nxtrk: 1call trkfmt
         \call cursoff
         \call keyoff
         1ld a, 0b7h
         1out (mscnt1), a
         1ld a, 0fah
         1out (mscnt1), a
         1ld a, 014h
         1out (mscnt1), a
         1in a, (mscio)
         1ld c, 0d3h
         1ld hl, FMTBLK
         1ld a, 0f4h
         1out (dskcmd), a
 1call waitdk
 1call cursun
 1call keyon
 1in a, (dskcmd)
 1and 044h
 1call nz, erprnt
 1inc d
 1ld a, 77
 1cp d
 1ret z
 1ld a, 05bh
 1out (dskcmd), a
 1call waitdk
 1in a, (dskcmd)
 1get status
and 010h : look for "seek error" bit
call nz, erprint : print registers if error
jp nxtrk : do another track

trkfmt formats 1 track of disk according to IBM 3740 format
with 256 bytes/sector.
INPUT: register d contains track No.

trkfmt:
1d h1,FMTBLK :
beginning of block storage to keep track info
1d b, 40 :
40 (decimal) bytes to be loaded
1d a,0ffh :
A = what to load (ff)
call bloa d:
writes 'B' bytes of (A), incrementing hl
1d b,6:
1d a,0:
call bloa d:
26 '0's
1d (h1),0fch:
fch = index mark
inc hl:
point to next location
1d b,26:
1d a,0ffh:
call bloa d:
26 'ff's
1d c,15:
want to write 15 sectors/track
call sect:
writes (C) sectors
1d b,0ffh:
1d a,0ffh:
call bloa d:
write a bunch of ff's till timeout
ret

end of trkfmt

sect writes (C) sectors worth of info into memory
INPUT: C is no. of sectors to write (destroyed)
D is track No. (kept)

sect:
1d b,6:
bload writes (B) bytes of (A), incs hl
1d a,0:
call bloa d:
1d (h1),0feh:
inc hl:
1d (h1),d:
inc hl:
1d (h1),0:
inc hl:
call secno:
1d (h1),a:
inc hl:
1d (h1),1:
inc hl:
1d (h1),0ff7h:
inc hl:
1d b,11:
1d a,0ffh:
call bloa d:
1d b,6:
1d a,0:
call bloa d:
1d (h1),0ffh:
inc hl:
1d b,0:
1d a,0e5h:
1d b,0:
setup for 256 bytes of e5
call bload  : actual data field loaded w/ a5
1d (h1),0f7h  : 2 CRC's
inc hl  :
1d b,27  :
1d a,Offh  :
call bload  : 27 ff's
dec c  : next sector
jp nz,sect  : do it again
ret  : else done, return.

end of sect
*************************************

secno returns a sector No. in A, given a sequence no. in C
and a track no. in D.
INPUT: reverse sequence No. in C (kept)
      track No. in D (kept)

secno:
push h1  :
1d a,d  : track No.
and 1  : look at lowest bit
jp z,even  : odd or even?
1d b,0  : if odd, look at oddlist
1d h,oddlist  : sequence for odd tracks
add h,bc  : offset by sequence No.
1d a,(h1)  : get that value
pop h1  : restore
ret  : done
even:
1d b,0  : even track No.
1d h,evnlist  : SO USE even list
add h,bc  : add offset of seq. No.
1d a,(h1)  : get it.
pop h1  :
ret  :

oddlist: db 0fh,14,13,12,11,10,9,8,7,6,5,4,3,2,1,15
evnlist: db 0fh,14,13,12,11,10,9,8,7,6,5,4,3,2,1,15

end of secno
*************************************

blload loads (B) locations with(A) in memory beginning
with (hl), incrementing hl as it goes. bl=0
blload:
1d (h1),a  :
inc hl  :
dec b  :
jr nz,blload  :
ret  :
end of blload
*************************************

restor:  : sets disk to track 0
push af  :
1d a,0bh  : restore, slowest stepping speed, loads head
out (dskcmd),a  : write to command register
call waitdk
in a,(ds:cmd)
and 01h
call nz,erprnt
ld a,000h
out (ds:cmd),a
pop af
ret

************************************************

erprnt:
push af
call pthxch
ld a,h
call pthxch
ld a,l
call pthxch
ld a,c
call pthxch
ld a,d
call pthxch
ld a,e
call pthxch
inc sp
inc sp
ex (sp),hl
ld a,h
call pthxch
ld a,l
call pthxch
ex (sp),hl
dec sp
dec sp
ret

************************************************

Dskdly delays for 7 x YY microseconds at 2 mhz,
or 3.5 x YY at 4 mhz.
dskdly:
push af
ld a,16
This is "YY"
dly1:
dec a
jp nz,dly1
pop af
ret

************************************************

RAM Definitions
dataorg 04000h
addr: dw 0h
keyflg: db 0h
kbfptr: db 0h
kbwptr: db 0h
curse: dw 0h
cstat: db 0h
ccchar: db 0h
dskrdy: db 0h
timfig: db 0h  ; timer flag
strtlc: dw 0h  ; starting memory location for transfers to IO
endloc: dw 0h  ; end memory location, or number of blocks for IO
tnode: dw 0h  ; INODE for disk or tape
tempt: db 0h  ; just a utility spot
dskid: db 0h  ; the loaded disk ID
trflg: db 0h  ; tape read flag
; tpbuf: db 0h  ; single byte tape buffer
danged if I know what that is!
; kbdbuf: ds 64  ; keyboard buffer (64 bytes)
rwflg: db 0  ; disk read/write flag.
crtmod: db 0  ; Mode of CRT
rowmo: db 0  ; row address variable, for direct cursor add.
Bios
BIOS PROGRAM
FOR Z-80 MICROPROCESSOR
by
L. A. TOMKO

: equates:

sp_ equ 020h
nl_ equ 0Dh
amp_ equ 040h
ESC equ 01h
ESCMOD equ 1
INSMOD equ 8
RMNOD equ 3
COLMOD equ 4
spbase equ 0Affh
tau equ 0Fah
crtrws equ 0800h
end eq crtrws + (2 * crtrws) - 1
hibot equ hibot/0100h
botilf equ crtrws + 080h
uplift equ crtrws + 080h
scdline equ crtrws + 0100h
botilf equ botilf + 1
kbsze equ 31
keyprt equ 0D6h
kbvct equ 0h
pm0d0 equ 0fh
pm0d1 equ 04fh
enable equ 083h
disable equ 3
dsktq equ 03h
dskc1 equ 02h
dsktrk equ 01h
diskctl equ 00h
DSKRST equ 00h
SEEK equ 01eh
UNLOAD equ 012h
RESTORE equ 0ah
WRK equ 044h
DNA equ 0Ch
WRST equ 0ah
RDST equ 080h
keybd equ 044h
misc1 equ 0D7h
misc2 equ 0D5h
RAM equ 02f
ROM equ 03f
t2 equ 0F6h
FMTRBLK equ 06000h
CPLDST equ 0000h

: org 0A100h
: starts at A100, which is supposedly safe
: reboot: d1
: in 2
: 1d hl,crtrws
: interrupt mode for Z-80 peripherals
: number of 2-byte words in the CRT RAM

Note: The handwriting is not clear, but it appears to be a comment or note related to the BIOS program.
ld sp,endcrt : last location of the CRT RAM
ld de,02020h : two ASCII 'blank's, one in each byte
           : writes two 'blanks' to consecutive CRT locations
clear: push de : looking for hl=0 to stop clearing CRT RAM
dec hl : test upper byte first
ld a,h : to make the flags appear
or a : if both are zero, we are done with clearing CRT
jp nz,clear : certainly not through if upper byte is > 0
ld a,l : now look at the lower byte
or a
jp nz,clear

Enable the pio chip and the interrupts

ld a, pmod1
out (keyprt),a : pmod1 = 4f, sets the pio to mode 1 (input)
out (keyprt),a : keyport is the control port for keyboard pio
ld a,kbvctr
out (keyprt),a : the interrupt vector for the keyboard (=0h)
ld a, enable
out (keyprt),a : enable = 083h, enables port interrupt
out (keyprt),a
in a,(keybrd) : do one read to set 'ready' output.

Initialize 1 ms timer.

ld a,07h
out (0f4h),a : sets timer mode, non-interrupting, prescale/16
ld a,0fah
out (0f4h),a : f4 is ctc channel 0.
ld a,08h
out (0f4h),a : time constant of 250 counts = 1 ms total
out (0f4h),a

out (0cfh),a : Set single density for floppy controller

Initialize PIOB, Port B for miscellaneous interrupts, including disk

ld a, 0cfh
out (mscntl),a : Mode 3 - bit I/O
out (mscntl),a : PIOB, Port B control
ld a,1
out (mscntl),a : DO only is input - all else = outputs
ld a,037h
out (mscntl),a : disable interrupt, active high, mask follows
out (mscntl),a : (This initialization is repeated in disk routines)
ld a,0fah
out (mscntl),a : Mask all but DO.
ld a,010h
out (mscntl),a : vector
in a,(mscio) : do a read to set 'ready' output.

misc. initialization

xor a
ld (keyflg),a : initialize keyboard flag = off
ld (kbpwptr),a : keyboard write pointer
ld (kbfpotr),a : keyboard read pointer
ld (cstat),a : start with cursor status = off
ld (diskid),a : default disk ID is 00
ld (crtnmd),a : CRT mode = normal
ld hl,crtmem : beginning of CRT RAM
ld bc, 0200h
add h, bc
ld (cursor), h
call cursorn
ld h, vectab
ld a, h
ld l, a

ld bc, 0fh
ld de, 0C057h
ld h1, xfrtbl
ldir

monitor:

ld sp, spbase
call dskdly
ld a, DSKRST
out (dskcmd), a
ld a, nl
ld b, putcrt
ret
ld h1, mon1
push h1
ret

mon1:

ld h1, mon2
push h1
ret

mon2:

ld h1, mon3
push h1
ret

mon3:

nop

:******** should jump to operating system from here.*****
jp COLDST 
: operating system entry (cold start).

:putcrt writes a character (found in a) to the crt
: much like a serial terminal.

putcrt:
push af
push bc
push de
push h1
call cursoff

ld c, a
ld a, (crtmod)
and 7
cp ESCMOD
jp z, escape
cp ROWMOD
jp z, row
cp COLMOD
jp z, column
ld a, c
cp ESC
jp z, setesc
ld a, (crtmod)

:offset for the initial cursor position
: start the cursor blinking
: the vector transfer table address is vectab (I hope)
: need upper byte in a to load reg. l.
: load the high byte of vector table

: Load the BIOS jump table.
: That's where it goes.
: That's where it comes from.

: reload the stack pointer.
: wait a few u-seconds for controller clear
: Reset controller, abort any commands.
: print a 'cr' on the crt
: enable the system interrupts.
: This next little operation clears peripheral devices
: that may have pending interrupts acknowledged but
: not cleared with a "reti" command. Each iteration
: clears only one device, so we will do three just
: to be sure!

: nop

: putcrt writes a character (found in a) to the crt
: much like a serial terminal.

:Turn the cursor off, which replaces the character
: at the cursor location and prevents interrupts.
: save the character in C for a while
: find state of CRT
: look at all but INSERT mode bit
: escape mode?
: Row mode?
: column mode?
: Now check for ESC character
: since not in ESCMOD, ROWMOD, or COLMOD.
: If Char = ESC, set ESCMOD.
cp INSMOD

jp z,insert
or a

and 0Ffh
jp m,special
ld a,c
ld h1,(curse)
ld (h1),a

ld a,1

ld 04fh
jp z,eoln

inc a
ld 1,a
ld (curse),h1

nullo: call curson
pop h1
pop de
pop bc
pop af
ret

eoln: ld de,031h
add hl,de
ld (curse),hl
ld de,botin
ld a,d
cp h
jp nz,nullo
ld h1,botift
ld (curse),hl
call scroll
jp nullo

special:
cp 080h
jp p,nullo
jp z,nullo

backsp:
cp 08h
jp z,backsp

cp 09h
jp z,tab

cp 0ah
jp z,lf

cp 0ch
jp z,clr

cp 0dh
jp z,cr
jp nullo

don't know what it is, so just ignore it.

backsp: ld h1,(curse)

ld a,1

ld a,1

ld 0fh
jp z,nullo

insert mode.
Any other mode is an
illegal CRT mode
restore character to A
mask off reverse video bit
check for special char (< 20h)
handle those separately
restore again
pointer to cursor
that's where we will write.
now, fiddle with the cursor.
lower byte of cursor address
that's eol for even rows
return and scroll
eol for odd lines
if not eol, just increment cursor address
don't worry about carry - never occurs in line.
update the cursor position
Now turn the cursor back on before departing
bye!
just wrote last char on line, move cursor to
beginning of next line. (by adding 31h)
update the cursor position
if cursor >= xdxx, we must scroll
just look at the upper byte
otherwise, we'll just return
scroll needed - first set cursor to bottom left
scroll moves everything up one, but leaves cursor.
handles special characters, like tabs, spaces, etc.
initially, if bit7 = 1 we will just ignore it.
backspace?
tab?
linefeed?
formfeed (clear screen)?
carriage return?
don't know what it is, so just ignore it.
backspace moves cursor back, does not erase.
stops backspacing at beginning of line.
cp 040h
jp z,nullo

dec a
ld 1,a
ld (curse),hl
jp nulio

: not at beginning of line, so decrement.

: put out spaces until cursor is left at even '8' mult.

: leaves cursor in same relative position.
: adding 80h to cursor position puts it strt. down
: however, if cursor >= 0300h, must scroll
: compare bottom of screen (high byte)
: if not below screen, reload cursor
: scroll won't change curse, so we just leave it alone
: gracefully return.

nxtline:
ld (curse),hl
jp nullio

: reload the cursor with the new value

: going to do 26 scrolls to clear the screen

: going to put cursor at top left
: graceful exit.

: return to start of line and do line feed
: look at the lower byte
: if greater than 7fh, we were on an odd line
: even line: return to 0,1) 
: moves the cursor back to start of current line 
: now do a line feed, note: routine returns 'cr'
: odd line: return cursor to xx80h.

escape:
ld a,(crtmod)
and INSMOD
ld (crtmod),a
ld a,c
cp 'E'
jp z,clr
cp 'K'
jp z,crlne
cp 'L'
jp z, insine

: previous character was and ESC
: clear all but INSERT mode bit
: look at the new character
: Home/Clear routine
: Clear to end of line
: insert new line at cursor row

These Sequences are modeled after
Hack/Em text.


```assembly
; blank line, moving lines below it down.
call crsbl

; clear carry for 16-bit subtract
sbc hl,de
ccf

; if so, just clear the line.
jp z,cirule

; next row up
ld hl,bolift-080h

; go to see if the activity is at (curse)
ld a,e
jp nz,ins1
ld a,d
jp b

; blank the row if at cursor

; delete line of cursor; moves lines below
; one up; blank line at bottom; cursor
; at beginning of deleted line.
call crsbl
ld d,h
```

```
cp 'M'
jp z,delne

; delete a line

; enter insert mode
cp 'Q'
jp z,exmod

; exit insert mode.

; delete character at cursor
jp z,delchr

; cursor row set
jp 'Y'

; ignore anything we don't understand
jr

; clear from cursor to end of line
cirule:
ld hl,(curse)

; get cursor position
ciruf:
ld a,l
cp 04fh
jp z,nulio
cp 0cfh
jp z,nulio
ld a,sp_
ld (hl),a
inc 1
jr ciruf

; clears even rows
ld a,sp_
ld (hl),a
inc 1
jr ciruf

; clears odd rows
ld a,sp_
ld (hl),a
inc 1
jr ciruf

; not done, so write a blank
ld (hl),a

; next...
```

```
; insert blank line at current cursor
ins1:
```

; cursor to beginning of line. -> HL

; clear carry for 16-bit subtract
sbc hl,de
ccf

; cur at bottom row?
jp z,cirule

; if so, just clear the line.
```

; next row up
ld hl,bolift-080h

; going to see if the activity is at (curse)
ld a,e
```

; if not, do some more
```
ld a,1
ld bc,080h
add h1, bc
ld bc, 80
call repeat
jp nullo

Next line below
80 columns
part of scroll: routine.

crsbol:
push af
ld h1, (curse)
dl a, 1
cp 0fh
jp p, crsb1
ld 0, 0
ld (curse), h1
pop af
ret

Puts cursor at start of current line,
returns position in HL

if > 7f, oddline
BOL for even line
reload it

crsb1:
ld 1, 080h
ld (curse), h1
pop af
ret

BOL for odd line = xx80

insmod:
dl a, INSMOD
ld (crtmod), a
jp nullo

Enter the insert mode

mvrow:
dl a, (crtmod)
or ROWMOD
ld (crtmod), a
jp nullo

enter row address mode
put rowmode bits in without changing insert mode

setesc:
dl a, (crtmod)
or ESCMOD
ld (crtmod), a
jp nullo

An ESC character has been received; set esc mode

exmod:
xor a
ld (crtmod), a
jp nullo

Exit insert mode — clear everything

delchr:
ld h1, (curse)
move text right of cursor left one slot.
delc1:
dl a, 1
cp 04h
jp z, delc2
end of line, even row
cp 0cfh
jp z, delc2
end of line, odd row
inc hl
ld a,(hl)
inc hl
ld (hl),a
jp delc1

; delc2:  
ld a,sp_
ld (hl),a
jp nullo

; row:
ld a,c
cp 25
jp m,row1
ld a,24

; row1:
ld '(rowno),a
ld a,(crtnod)
and INSMOD
or COLMOD
ld (crtnod),a
jp nullo

; column:
ld a,(crtnod)
and INSMOD
ld (crtnod),a
ld a,c
cp 80
jp m,col1
ld c,79

; col1:
ld de,080h
ld hl,uplit
ld a,(rowno)
or a
jp z,col2

; col2:
add hl,de
dec a
jp nz,col2

; col3:
ld a,c
add a,1
ld l,a
ld (curse),hl
jp nullo

; insert:
ld de,(curse)
ld h,d
ld a,e
cp 80
jp p,insert

; Inserts a character at cursor position
; Won't DVANCE BEYOND EOL
; > 80 --> odd line
ld 1,04fh
jp insrt2

insrt1:
ld 1,0cfh
: EDL for even line

insrt2:
: EDL for odd line
ld a,e
cp 1
jp z,insrt3
dec h
ld a,(hl)
inc l
ld (hl),a
dec l
jp insrt2

insrt3:
ld (hl),c
ld a,l
cp 04fh
jp z,insrt4
cp 0cfh
jp z,insrt4
inc hl

insrt4:
ld (curse),hl
jp null0

End of putcrt

;-----------------------------

; audcrt1:
push af
push hl
ld hl,crtm1
load message address
call patmng
ld a,(crtmod)
print it
ld 1,a
save the crt mode
xor a
clear the crt mode to normal
ld (crtmod),a
print the offending mode
ld a,1
call putcrt
pop hl
ret

; crtm1: db 'Illegal CRT mode',0

;-------------------------------------------------

; hexchr:
push af
push bc
ld b,a
save byte
and 0fh
look at lower nibble
if < 10, must be 0 - 9
jp 10
digit
must be > 10, so subtract 10, and add A
sub 10
add a,06h
's'
ld 1,a
that's the lower character
digit:
add a,030h
'0' ASCII
thigh:
  ld l,a
  ld a,b
  rrc a
  rrc a
  rrc a
  rrc a
  and 0fh
  cp 10
  jp m, dig2
  sub 10
  add a,061h
  ld h,a
  jp thru

dig2:  add a,030h
  ld h,a
  thru:
    pop bc
    pop af
    ret

******************************************************************************

curses:
  push af
  push bc
  push hl
  ld hl,(curse)
  ld a,(hl)
  xor 080h
  ld (hl),a
  ld a,(cstat)
  or a
  jp z,tturnon
  xor a
  a = 0
  ld (cstat),a
  clear cursor status.
  jp home
  return
  
  turnoff:
    ld a,01h
    ld (cstat),a
    set cursor status = ‘on’ (1)
  
  turnon:
    pop hl
    pop bc
    pop af
    ei
    Turn those interrupts back on!
  
  return

******************************************************************************
cursoff:
  push af
  push hl
  ld a, (cstat)
  or a
  jp z,blinkck
  ld hl,(curse)
  ld a,(hl)
  xor 080h
  ld (hi),a
  xor a
  ld (cstat),a
  Is the Blinker activated?
or a
jp z, donoff
ld a, 043h
out (0f7h), a
xor a
ld (blink), a
; and Zero the Blinker status.
donoff: pop h1
pop af
ret
byel

; End cursor

;UFFIXS

curson: push af
push h1
ld a, (cstat)
or a
jp nz, ckbink
ld h1, (curse)
ld a, (hi)
xor 080h
ld (hi), a
ld a, 1
ld (cstat), a
; Set the cursor status = ON.
ckbink: ld a, (blink)
or a
jp nz, donon
ld a, 0c7h
out (0f7h), a
ld a, tau
out (0f7h), a
ld a, 1
ld (blink), a
; Set blinker status.
donon: pop h1
pop af
ret

; End cursor

;UFFIXS

scroll: push af
push bc
push de
push h1
call cursonoff
ld bc, 050h
ld de, uplft
ld hi, scdine
call repeat
pop h1
pop de
pop bc
pop af
ret

;UFFIXS

repeat: ld dir
ld bc, 030h
add h1, bc
(de) < (hi), h1++, de++, bc-- till bc=0. Nifty!

; enough to get to start of next line.
push hl
ex de,hl
add hl, bc
ex de, hl
pop hl
ld bc, 050h
ld a, h
cp hibot
jp nz, ret
ld hl, bot1f
ld de, bot1f
ld (hl), 020h
ldir
ret

End repeat

keyinr:
push af
push bc
push hl
in a,(keybrd)
lc c,e
ld a,(kbwptr)
inc a
and kbfse
id(kbwptr), a
ld h1,kbdbuf
add a, l
ld h, a
ld (h1), c
ld a, l
ld (keyflag), a
pop hl
pop bc
pop af
el
ret

End keyinr

pstrng:
push af
strglp:
ld a,(hl)
cp 0
jp zn, strglt
call putcrt
inc hl
jp strglp
strglt: pop af
ret

End pstrng

pthxch:

Puts hex number to CRT as two ASCII characters
push hl
call hexchr
ld a,h
ld a,1
call putchr
pop hl
ret

End pthxch

ldsv:
ld a,5
ld (dsktry),a
ex de,hl
pop hl
ex (sp),hl
push de
ld de,15
ex de,hl
call ccdiv
inc e
ld d,1
ld c,h
ld a,(dskid)
inc dl
now track = d
ld c,6
ld a,0
out (dskcmd),a
out (dskdly),a
call dskdly
in a,(dskcmd)
Look at status
and 020h
look at Head Load bit.
jp nz,ldsvq
ld a,d
cp 77
jp p,errmsg9
out (dskdta),a
call dskdly
out (dskcmd),a
call waitclk
in a,(dskcmd)
get status
out (dskcmd),a
out (dskdly),a
wait a few microseconds
set status
load data register
out (dskdta),a
wait a few microseconds
and 018h
jp nz,ldstr
ldsvq:
pop hl
return with r/w address in hl
ret

errmsg5: ld hl,errmsg5
"Seek error"
call putstr
erdata: ld a,d
track
call pthxch
ld a,e
sector
call pthxch
jp reboot

msg5: db 'Seek Error',0
ermsg9: ld hl, msg9
    call pstrlen
    jr erdata
    ; "Track out of range"
    ; Also print track and sector.

msg9: db 'Track out of range ', 0

ldstry:
    ld a, (dsktry)
    dec a
    jd z, ermsg5
    ld (dsktry), a
    call restor
    jr ldsyu

; Try (dsktry) times to seek track
; Give up after (dsktry) tries.
; backup to track 00

End ldsyu

; *************************************************
; save:
; Writes one block to disk
; given address in HL, and Tnode second on stack,
; Return address
pop bc
pop de
push bc
push de
    ; Save the return address
    ; Give next routine Tnode in stack
    ; Common disk ID program unrolls Tnode and gets
    ; to right track on right disk (or aborts)
    ; 5 tries to write
ld a, 5
    ; save start address in case of abort
ld (dsktry), a
push hl
    ; save curoff
    ; Don't want interrupts during write,
    ; don't allow keyboard to interrupt
    ; enable PIO int; 'or' logic; active high; mask follows
    ; Mask all but D0.
    ; Vector --> dskout interrupt handler.
call curoff
    ; E should contain sector number
    ; put that in disk sector register
    ; clears PIO ready signal.
    ; Wait a few microseconds
    ; Prevents B from decrementing to zero in 'dskin'
    ; set C to controller data port
    ; write sector command (0a8h)
    ; to the command register
    ; Now just sit back and wait for disk to finish
    ; Let 'er blink again
    ; mask the write-pertinent bits
    ; print write-error message if any bad bits
    ; restore stack
    ; otherwise, return
    ; ret
    ; should contain Tnode
    ; write it to CRT
ld a, e
    ; lower byte
ld (dskxch), a
    ; msg7: db 'Disk write error', 0
savtry:
  ld a,(dsktry)
  dec a
  jp z,ermsg7
  ld (dsktry),a
  pop hl
  push hl
  recall the start address
  save it again, in case another retry

End save

keyon:
  push af
  ld a,enable
  out (keyprt),a
  pop af
  ret

Turns on the keyboard

keyoff:
  push af
  ld a,disable
  out (keyprt),a
  pop af
  ret

Turns off keyboard

End keyon/keyoff

iskout:
  Disk output interrupt handler.
  NOTE: this has to be FAST!
  INPUTS: HL contains the address to be written
  B contains the disk controller data port.
  S should be written to a high value to prevent
  it from decrementing to zero, which would
  set the "Z" flag and screw up the dskwait
  routine.

outi (HL) > (C); inc HL; dec B
in a,(mscnt1) clears the PIO interrupt.
el reti

end dskin

load:
  Reads a sector to memory specified in HL
  from disk at Tnode, contained in stack
  push bc
  pop bc
  push de
  pop de
  call ldav
  ld a,5
  ld (dsktry),a
  push hl

load1:
  Save start address in case of abort
  call cursor
  call keyoff
  ld c,0d3h
  Setup C for the "diskin" interrupt handler.
ld a, 0B7h
out (mscnc1),a
ld a, 0Feh
out (mscnc),a
ld a, 012h
out (mscnc1),a
in a, (mscl0)
l d b, 0FFh
ld a, RDSCT
out (dskcmd),a
call dskdly
ld d, 01Ch
jp nz, ldtry
pop hl
rest
hl
else done - return

ermsg6:

call erprnt
ld h, msg6

call pstrng
ld a, d

call phxch
ld a, e

call phxch
jp reboot

msg6: db 'Disk read error. Track/sector = ', 0

ldtry:

ld a, (dsktry),

dec a

jp x, ermsg6

ld (dsktry), a
call restor
ld a, d

out (dskcnta), a
call dskdly
ld a, SEEK
out (dskcmd), a
call waitdk
in a, (dskcmd)
and 018h
jp nz, ldtry

pop hl
push hl

jp load!

end load

----------
dskin:

Interrupt handler for disk reads.

NOTE: this routine must go FAST!

INPUTS: - ML contains the start address to be loaded to. It will be incremented each call.
- C contains the input port address.
- B will be decremented each time, so flag register will be affected when
B becomes 0. This may interact with calling
program. Set B to 'FF' before starting to
minimize this.

int
nop
input (C) and inc HL. (16 cycles)
(4 cycles) This is needed temporarily to
prevent bad memory read of 'ff'.
esi
(4 cycles)
reti
(14 cycles) resets PIO without reading it.

end diskin

*******************************

waitdk:
push af
ld a,16
 setup 43 microsecond wait, so controller
 status will be valid after command.

waitdk1:
ld a,16
 dec a
jp nz,waitdk1
 This loop kills 7 microseconds per pass
at 2 mhz, 3.5 at 4 mhz system clock.

waitdk2:
ld b,0ffh
 B is decremented (but not tested) in disk
interrupt routines (diskin, diskout). We
write B to ff to prevent a 'z' condition
in the flag register in case the interrupt
occurs during the 'and' operation. We may
therefore miss the resetting of the 'busy'
status bit for one loop, but no matter.
in a,(dskcmd)
 now start looking at controller status register
and 1
jp nz,waitdk2
 loop if still busy
pop af
ret

End waitdk

*******************************

ccdiv:
ld b,h
ld c,l
ld a,d
xor b
push af
ld a,l
or a
call a,ccdenneg
ld a,h
or a
call a,ccbcneg
ld a,16
push af
ex de,hl
ld de,0

ccdiv:
add hl,hl
cc call cccdiv1
ld x,ccdiv
cc call cccdiv2
cc call cccdiv2
cc call cccdiv2
cc call cccdiv2
cc call cccdiv2
ld a,1
or 1
ld l,a
ld h,a
sub c
ld e,a
ld a,d
sbc a,b

Excluding comments, the code contains instructions for manipulating and processing data, with a focus on bitwise operations and memory management. The comments provide explanations for various parts of the code, including the purpose of certain instructions and the context in which they are used. The code also includes a function for dividing registers and handling the quotient and remainder, with additional functions for managing the stack and executing conditional jumps.
ld d,a
ccdiv2: pop af
dec a
jp z,ccdiv3
push af
jp ccdiv1

ccdiv3: pop af
ret p
call ccdeneg
ex de,hl
call ccdeneg
ex de,hl
ret

ccdeneg: ld a,d
; negates the integer in DE
cpl
ld d,a
ld a,e
cpl
ld e,a
inc de
ret

ccbcneg: ld a,b
; negates the integer in BC
cpl
ld b,a
ld a,c
cpl
ld c,a
inc bc
ret

ccdel: ld a,e
; Rotate DE left one bit
rla
ld e,a
ld a,d
rla
ld d,a
or a
ret

ccmpbcd: ld a,e
; compare BC to DE
sub c
ld a,d
sbc a,b
ret

org 0A800h
vectab:
dw keyinr 00 > keyboard interrupt
dw vcterr 02 > tape output - not implemented yet
dw vcterr 04 > DMA controller - not supposed to interrupt
dw vcterr 06 > tape input - not implemented yet.
dw vcterr 08 > CTC channel 0 - shouldn't interrupt.
dw vcterr 0a > CTC channel 1 - not defined yet.
dw vcterr 0c > timer channel.
dw curses 0e > cursor interrupt.
dw vcterr 10 > PIO interrupt - formerly DISK controller.
dw dskin 12 > PIO for disk input operations.
dw dskout 14 > Same PIO for disk output operations
vcterr:          routine to handle strange vectors from interrupts.
push h1
ld h1, msg8    "unknown interrupt vector"
call pstrlen
pop h1
ei
ret

msg8: db 'Unknown interrupt vector', nl, 0

end vectab

*****************************

ERROR MESSAGES

ermsg4: ld h1, msg4 'wanted disk/Found disk: '
call pstrlen
call pthxch desired disk was in A
ld a, c
load disk in C
ld a, c
j p monitor

msg4: db 'wanted disk/Found disk: ', 0

*****************************

format: A disk formatting program for 8'
single density single side disk.

format:
call restor
ld d, 0
D will contain the track No.
call trkfat
ld a, 0B7h
out (mscntl), a
ld a, 0feh
out (mscntl), a
ld a, 014h
out (mscntl), a
in a, (mscio)
lc 0, 03h
ld h1, FMTBLK
ld a, 0F4h
out (dskcad), a
ld a, 044h
out (dskcad), a
call waitdk
call curson
in a, (dskcad)
call dskdly
ld a, 077
inc d
ld a, 05b
out (dskcad), a
ld d, 0
ret z
ld a, 05b
out (dskcad), a
call waitdk
in a, (dskcad)
call dskdly
and 010h

look for "seek error" bit
ld (h1),0f7h ; 2 CRC's
inc h1
ld b,27
ld a,0ffh
call block
dec c ; next sector
jp nz,sect ; do it again
ret ; else done, return.

end of sect

*************************

secno returns a sector No. in A, given a sequence No. in C
and a track No. in D.
INPUT: reverse sequence No. in C (kept)
track No. in D (kept)

secno:
push hl ; track No.
ld a,d
and 1 ; look at lowest bit
jp z,even ; odd or even?
ld b,0 ; if odd, look at oddlist
ld 'h1,oddlist ; sequence for odd tracks
add hi,bc ; offset by sequence No.
ld a,(hl) ; get that value
pop hl ; restore
ret ; done

even:
ld b,0 ; even track No.
ld 'h1,evenlist ; 50 USE even list
add hi,bc ; add offset of seq. No.
ld a,(hl) ; get it.
pop hl
ret

oddlist: db 0ffh,8,16,7,14,6,13,5,12,4,11,3,10,2,9,1
evenlist: db 0ffh,11,3,10,2,9,1,8,16,7,14,6,13,5,12,4

end of secno

*************************

block loads (8) locations with(A) in memory beginning
with (h1), incrementing hi as it goes, bl=0
block:
ld (h1),a
inc hi
dec b
jr nz,block
ret

end of block

*************************

<<NOTE: Would prefer to unload head first to clear dust,
etc, but to do so would turn off drive as it is
now configured. This is a good area for future
modification. 'UNLOAD' has been defined as a
'SEEK' without head load or verify. This can
be used to unload head in place, by simply
loading the data register with the contents
call nz, erprint ; print registers if error
jp nxtrk ; do another track

******************************

trkfmt formats 1 track of disk according to IBM 3740 format
with 256 bytes/sector.
INPUT: register d contains track No.

trkfmt:
ld hl,FNTBLK   : Beginning of block storage to keep track info
ld b,40       : 40 (decimal) bytes to be loaded
ld a,0ffh      : A = what to load (ff)
call biolod    : writes 'B' bytes of (A), incrementing hl
ld b,6         :
ld a,0         :
call biolod    : 6 'O's
ld (hl),0ffh    : fch = index mark
inc hl         : point to next location
ld b,26        :
ld a,0ffh       :
call biolod    : 26 'ff's
ld c,15        : want to write 15 sectors/track
call sect      : writes (C) sectors
ld b,0ffh       :
ld a,0ffh       :
call biolod    : write a bunch of ff's till timeout
ret

******************************

sect writes (C) sectors worth of info into memory
INPUT: C is no. of sectors to write (destroyed)
       D is track No. (kept)
sect:
ld b,6       :
ld a,0       :
call biolod  : biolod writes (B) bytes of (A), incs hl
ld (hl),0ffh  : ID address mark
inc hl       :
ld (hl),d     : track no.
inc hl       :
ld (hl),0     : side no. (always 0)
inc hl       :
call secco   : returns a sector no. in A, given C and D
ld hl,a       :
inc hl       :
ld hl,1       : sector length (1 -> 256 bytes)
inc hl       :
ld (hl),0ffh  : f7 writes 2 CRC's
inc hl       :
ld b,11       :
ld a,0ffh     :
call biolod  : 11 ff's
ld b,6        :
ld a,0        :
call biolod  : 6 'O's
ld (hl),0ffh   : data address mark
inc hl       :
ld b,0        : setup for 256 bytes of e5
ld a,0e5h     :
call biolod  : actual data field loaded w/ e5
of the track register before issuing. >>

: restor:
push af
ld a,DSKRST
out (dskcmd),a
call dskdly
ld a,RESTORE
out (dskcmd),a
call waitdk
in a,(dskcmd)
call dskdly
and 010h
call nz,erprnt
jp nz,erprnt
ld a,0d0h
out (dskcmd),a
call dskdly
pop af
ret

: Dskdly delays for 7 x YY microseconds at 2 mhz.
or 3.5 x YY at 4 mhz.

: dly1:
push af
ld a,16
inc a
jp nz,dly1
pop af
ret

: erprnt:
push af
call pthxch
ld a,h
call pthxch
ld a,l
call pthxch
ld a,b
call pthxch
ld a,c
call pthxch
ld a,d
call pthxch
ld a,e
call pthxch
inc sp
inc sp
ex (sp),hi
ld a,h
call pthxch
ld a,l
call pthxch
ex (sp),hi
dec sp
dec sp
pop af
ret

sets disk to track 0
reset the controller
wait for that to take effect
restore, load head, verify
write to command register
wait for controller to interrupt
wait a few microseconds
look for "seek error" bit
print error if found
Give up after printing.
reset controller
wait a few microseconds
this is "YY"
prints registers and return address
preserve a register
prints contents of A register
print H and L registers
print B register
print C register
print D register
print E register
look at return address
put it in hi and save hi in stack
print high byte of return address and low byte
restore the return and hi
restore the STACK POINTER
conin:

Reads characters from the keyboard, checks for special characters, returns these to the operating system.

INPUTS: none.
OUTPUTS: character in HL

ld a,(keyflag)
or a
jr z,conin
ld a,(kbfptr)
in c
and kbsize
ld (kbfptr),a
ld b,0
ld h1,kdbuf
add h1,bc
ld a,(kbfptr)
cp c
jr nz,conini
xor a
ld (keyflag),a

conini:

ld a,(h1)
and 07fh
ld l,a
ld h,0
jr z,nline
cp 0
jr z,eofile
end of file.
cp 013h
jr z,conin
ignore it.
cp 011h
war start.
cp 018h
cold start.
ret
eofline:

ld 1,0dh
translate to carriage return
ret
eofile:

ld h,080h
translate to 8000h
ret

end of conin

conout:

transmits a character to the CRT.
Actually, it interfaces with putcrt, which is a software package of considerable size that does the work.

INPUTS: character in L
OUTPUTS: none.

ld a,l
cp nl
jr z,Conout
get character into A.
if newline, check for restarts.
call putcrt:
write character in A to CRT.
ret
call chkio
ld a,nl
call putcrt
ret

end conout

chkio:
ld a,(08fd6h)
or a
ret nz
ld a,(keyflg)
or a
ret z
ld a,(kfbptr)
inc a
and kbsze
ld c,a
ld b,0
ld hl,kdbuf
add hl,bc
ld a,(hl)
and 0f7fh
cp 01h
jp z,0c0003h
cp 018h
jp z,0c0000h
cp 013h
jr z,hold
ret

hold:
xor a
ld (keyflg),a
ld a,c
ld (kfbptr),a
hold1:
ld a,(keyflg)
look for another keystroke
or a
jr z,hold1
ret

end chkio

BIOS jump table. The following block is relocatable, and
should be loaded at 0c057h.
xfrtbl:
jp conin
jp conout
jp load
jp save
jp format

..................................................
RAM Definitions

addr: dw Oh
keyfig: db Oh
kbfptr: db Oh
curse: dw Oh
blink: db Oh
cstat: db Oh
dskrdy: db Oh
timfig: db Oh
strloc: dw Oh
endloc: dw Oh
tnode: dw Oh
temp: db Oh
dskid: db Oh
trfig: db Oh
tpbuf: db Oh
stfig: db Oh
kdbbuf: ds 64
rwfig: db 0
crtmod: db 0
rown: db 0
dsktry: db 0

place to store the address from keyboard
flag set when keyboard is depressed
offset for reads from keyboard buffer
keyboard entry pointer
address of cursor
Blinker status (on = 1, off = 0)
status of cursor (on = 1, off = 0)
disk ready flag
timer flag
starting memory location for transfers to IO
end memory location, or number of blocks for IO
INODE for disk or tape
just a utility spot
the loaded disk id
tape read flag
single byte tape buffer
danged if I know what that is!
keyboard buffer (64 bytes)
disk read/write flag
Mode of CRT
Row address variable, for direct cursor address
Number of tries to read/write disk (usually 5)