# HEXDOS

The smart disk operating system
for Ohio Scientific computers

# INTRODUCTION

This software has been thoroughly tested and is believed to be reasonably bug-free. In spite of this, somewhere in the many scores of man-hours and the thousand-odd lines of assembly code involved in this project, there is guaranteed to be at least one bug lurking.

Other than the real-time clock, the hardware modifications listed in appendix D are not intended to imply any approval from OSI. Hardware changes are not essential to use this disk operating system, but you will find your machine more convenient to use after making the changes.

At this writing, HEXDOS is in version 4.0. This version is compatible with the standard OSI monitor ROM as well as the C1S and C1E ROMs. The editing features provided by these replacement ROMs take precedence over those provided by HEXDOS, and all features are available directly from the keyboard. New disks of this version or future versions are available from The 6502 Program Exchange to prior purchasers of HEXDOS for $10.

Questions, comments, suggestions, errors, etc. should be addressed to the author:

Steven P. Hendrix
Route 8, Box 81 E
New Braunfels, TX 78130

Distributed by The 6502 Program Exchange
2920 West Moana
Reno, NV 89509

## CONTENTS

## GETTING STARTED

We'll skip the usual congratulations on your purchase and get right down to business. Turn on your system, press BREAK, insert the HEXDOS diskette, and press D. HEXDOS will boot up using parts of the cold start routine, including the MEMORY SIZE and TERMINAL WIDTH functions, which run normally.

This manual will assume that you are familiar with BASIC-in-ROM and in particular with the requirement to press RETURN after typing each line. If not, please read the manuals which came with your system and become familiar with BASIC-in-ROM before proceeding.

After loading HEXDOS as in the paragraph above, you can treat the system as if you were using BASIC-in-ROM, because in fact you are. HEXDOS gets its incredible compactness by using the power built into the system wherever possible.

Please avoid the USR function until you read the section describing its special powers, and use LOAD and SAVE only as described below until you read the appropriate sections. If you leave the write-protect tape on the disk, you need not fear damaging it by tinkering with the new commands at your fingertips. Once you remove this protection you are on your own.

To get a directory, type LOAD/ . This will load the disk directory as a BASIC program which you can LIST. To load a program, for instance FORMAT, type LOAD "FORMAT". To save a program after editing, type SAVE. To create a new copy and call it NAME (for example), type SAVE "NAME".

## SPECIAL KEYS

All normal key functions of BASIC-in-ROM are available under HEXDOS, along with the following additional features. Any word enclosed in [ ] refers to a single key. Where two keys are in brackets, hold the first key while pressing the second key.

| | |
|---|---|
| [ctrl] | Suspend output, wait until the key is released. |
| [rub out] | Move the cursor nondestructively backward. |
| [esc] | Move the cursor nondestructively forward. |
| [shift return] | Allows editing (see below). |
| [repeat] | Break a BASIC program (identical to [ctrl C]). If change is made as in appendix D, repeats the last command entered in immediate mode. |
| [ctrl C] | Instant screen clear anytime you can type characters on the keyboard. Breaks a BASIC program otherwise. (Note that output is suppressed while you hold the ctrl key; the screen-clear character will not be issued until you release the ctrl key.) To clear the screen from a BASIC program, PRINT CHR$(3); |

Any key functions provided by alternate ROMs supercede those provided by HEXDOS.

## EDITING (inactive with C1S)

HEXDOS adds the capability to edit a line which you are entering or have already entered without retyping the entire line.

To edit a line such as line 40, simply list the line but hold the shift key while pressing return (or type ctrl-shift-M if you have the C1E ROM). That is, type

```
LIST 40 [shift return]
```

Line 40 will list on the screen and be placed in the edit buffer just as if you had typed the line but had not yet pressed [return]. Now you may edit the line just as you may when first typing it by using [rubout] to backspace the cursor and [esc] to move the cursor forward. Simply type over any part of the line you wish to change. Note that you must use [esc] to move the cursor to the end of the line and then press [return] to enter the line when you have finished editing it.

## LOAD

HEXDOS adds the following forms of LOAD. Items enclosed in [ ] refer to an item that should appear there (don't type the brackets).

| | |
|---|---|
| [aex] | Any arithmetic expression valid in BASIC. A number such as 5, a variable such as I, a subscripted variable such as A(I), or even an expression such as A(I)+LOG(SIN(Q/2))/43.787 would be valid. The expression must result in a value in the range -32768 to 32767. |
| [filename] | Any valid filename. (See SAVE for rules.) |

LOAD/ Load the directory. Use LIST to view it after loading.

LOAD [filename]
Load [filename] as a BASIC program. Example: LOAD "FORMAT". If [filename] starts with $, it refers to a machine language file containing load address information as noted in the USR section of this manual. Note that using LOAD "$name" in a program will clear all variables. RUN [filename] loads a BASIC program from disk and runs it.

LOAD *[aex],[filename]
Open [filename] as the data file specified by [aex]. Example: LOAD *5,"MYFILE". Each use of this form allocates a 2K byte buffer from BASIC's string space. A call to FRE(X) deallocates all file buffers (see Appendix F).

LOAD #[aex1],[aex2]
Directly loads track number [aex1] into the 2K bytes beginning at [aex2]. *Use with caution,* preferably with nothing in memory which is not saved. Example: LOAD #1,8192 .

LOAD! On a dual drive system, selects the drive which is currently idle and homes its head to track 0. A program may detect which drive is active as in this example:

```
IF (PEEK(49152) AND 64) THEN PRINT "A IS ACTIVE"
```

This quantity is true if drive A is selected and false if drive B is selected. Drive B must be initialized with LOAD! before it is used. Track number 128 is drive B track 0, etc.

## SAVE

SAVE functions similarly to LOAD with a few exceptions. See the description of LOAD for general comments. SAVE in its normal function dealing with cassette tape is unnecessary under HEXDOS. NULL10:LIST#2 will perform the same function as the old SAVE:LIST without writing extraneous data on the tape. (See also LIST.)

SAVE Save the current BASIC program to the same tracks it came from. Note that the same disk must still be in the same drive, since SAVE will write over whatever is on the corresponding tracks of another disk.

SAVE [filename]
Check to see that [filename] does not already exist, create a new file named [filename] large enough to hold the current program, and save the program in the new file.

SAVE &[filename]
Save the current program in the existing file [filename]. The old contents of [filename] are lost. Example: SAVE &"TEST"

SAVE *[aex]
Close output file [aex] and write the last buffer to the disk (generates F ERROR if used on an input file).

SAVE #[aex1],[aex2]
Save the 2K bytes starting at [aex2] on track number [aex1]. *Use with extreme caution, preferably on an empty disk!*

### Filenames

A filename may be any valid string expression. If you type the name directly in a line, this means you must enclose it in quotes. You may also specify a filename with a string variable or a concatenation of strings. A filename which starts with $ will be interpreted as a machine language file with load address information (see USR, CREATE, and DISK FORMAT).

## INPUT, PRINT, AND LIST

Input and output are controlled by HEXDOS using the forms INPUT#*n*, PRINT#*n*, and LIST#*n*, where *n* selects the appropriate device from the table below. For instance, to list a program to a printer, simply LIST#1 . This will list the program but send the listing to the printer rather than the screen. Note that LIST by itself does not generate a carriage return after printing the last line of the program. Most printers do not print a line until they receive a CR, so you may need to PRINT#1 after a LIST#1 to see the last line.

INPUT can get its data from any device listed below, and PRINT can send its output to any device. If any of these verbs are used without #, device 0 is assumed. Thus programs dealing only with the keyboard and screen will run without modification.

Input devices

| | |
|---|---|
| 0 | keyboard |
| 1 | reserved for expansion |
| 2 | 6850 ACIA (tape interface, printer, or modem depending on your hardware) |
| 3 | reserved for expansion (modem) |
| 5–25 (odd) | disk input files |

Output devices

| | |
|---|---|
| 0 | video screen |
| 1 | parallel printer port at $D900 |
| 2 | 6850 ACIA |
| 3 | reserved for expansion (modem) |
| 4–24 (even) | disk output files |

OSI made an error in the routine which prints error messages which causes the second character to be printed as a graphic character (bit 7 is a 1). HEXDOS prints the correct two-letter error identifier by masking off bit 7. If you wish to print graphic characters (decimal 128 thru 255), POKE 227,255 to disable this feature. POKE 227,127 will restore it.

## USING DISK DATA FILES

A HEXDOS data file may be visualized as a video screen which may be written to and read from. A PRINT statement writes the same characters to a disk file as it would write on the screen. This includes control characters such as carriage return and line feed, which are not normally visible on the screen. An INPUT statement accessing a disk file will react exactly as if you typed the characters on the keyboard.

The following short example should help to clarify disk data file use for those who are not familiar with the concepts involved. This example will assume a file named MYDATA already exists (use the CREATE utility to make such a file one track long for this example). First, we must associate a file number with the file MYDATA and prepare it for writing. This process is commonly referred to as "opening" the file. HEXDOS uses the BASIC verb LOAD for this purpose:

```
LOAD *4,"MYDATA"
```

This sets the disk file for output (note the even file number) and "clears the screen" so that writing will start at the beginning of the file. Print a line into the file just as you would on the screen:

```
PRINT #4,"HELLO THERE!!"
```

You may also print numbers. To print several numbers on one line so that they may be read properly, you must print commas between them just as you would if you were typing on the screen:

```
A=12.5:B=23
PRINT #4,A;",";B
```

After writing to the file, it must be properly closed to insure that all data is correctly entered on the disk. HEXDOS uses the BASIC verb SAVE to close a file:

```
SAVE *4
```

Now the data file is actually on the disk, and may be read by any program.

Since BASIC does not permit using the INPUT statement from the keyboard, we will have to create a short program to show how to read from the data file we just created:

```
NEW
10 LOAD *5,"MYDATA"
20 INPUT #5,A$
30 PRINT A$
40 INPUT #5,X,Y
50 PRINT X,Y
```

Line 10 opens the file as before, but uses an odd file number to designate an input file. Line 20 reads the first line of the file as if you had typed the line on the keyboard. Line 30 shows the results. Line 40 reads the next line, on which we printed the numbers 12.5 and 23 previously. Line 50 shows the results. Since we are not changing the file, there is no need to close it.

You may output on the same line with several different PRINT statements by ending them with a semicolon, just as you can on the screen. For the most compact data files, use NULL 0 to prevent BASIC from placing nulls after each line (they take up space but do not appear in the data when it is read back). HEXDOS selects this setting during boot-up, so it need not be changed unless you change the setting elsewhere.

## RANDOM-ACCESS DATA FILES

There are two types of data file organization. The file we worked with in the last section was a "sequential" data file. A sequential data file can be visualized as one long string of characters that are always read from or written to sequentially. In other words, the third character in a sequential file is always read after the second character and before the fourth. BASIC does input and output on such a file as if those same characters came from the keyboard or went to the screen.

A "random-access" file, on the other hand, can be seen as a set of numbered strings of characters. Each of these numbered strings is called a "record." The data contained in each record is stored sequentially, as before. The thing that distinguishes a random-access file is that records do not have to be written or read in any particular order. A program might store data in record 5, record 2, and record 99, then retrieve data from record 43 and record 1. If you tried to do this with a sequential data file, the computer would have to read or write to all intervening records before operating on the record it wished.

The HEXDOS disk contains a program called RANFILE to demonstrate random-access data files. The subroutine contained in RANFILE sets the file pointer to a specific record whose number is given by the user in the variable ZP. If your program needs to access record number 100, for example, you would set ZP=100, GOSUB the appropiate routine in RANFILE, and then INPUT or PRINT to the record as desired.

In a sequential data file, individual items are separated by commas, colons, or carriage returns. The items may be any length, which adds flexibility but makes it difficult to locate a specific item. In a random-access file, items within a record may still be of varying lengths, but records must be a fixed length to allow the computer to find the beginning of the record. RANFILE assumes each record is 64 characters long (including control characters). This length may be changed by changing the value of ZL in line 63080.

Remember that BASIC prints to a sequential or random file just as it does to the screen; that is, it prints spaces between items separated by commas, and prints a carriage return, line feed, and nulls at the end of every print statement that does not end in a semicolon. During input, BASIC continues to call for characters until getting a carriage return and then analyzes the line it has received. A record being read in a random file continues into the next record if necessary.

For further information, list RANFILE.

## USR

The USR function is automatically defined for a number of uses by HEXDOS. The usual way to access it from BASIC is T=USR(X) or PRINT USR(X), where X selects the function:

| Value of X | Function |
|---|---|
| 256 and up | Tone generator (you need to connect a speaker as noted in Appendix D). X is 256*length+pitch. |
| 0 thru 255 | Input a character from device X. This form may not be used in a PRINT statement or immediate mode. |
| -1 | Return a number identifying the disk error type when used with a BASIC error handling routine (see Appendix A). |
| -4 thru -2 | Return the value of one of the three bytes of the real-time clock. USR(-4) returns the least significant byte, while USR(-2) returns the most significant byte. |
| -5 | Jump to a machine language routine at the location specified by an arithmetic expression following USR. That is, T=USR(-5) 0 would jump to location $0000 (warm start), while T=USR(-5) -1024 (i.e., 64512) would jump to the disk boot routine at $FC00. Use with caution. Once you pass control to a machine language program, you lose the protection provided by BASIC. It may erase your program, write on the disk, or just go away and not come back. You may recover from the latter by doing a warm start with [break] followed by W. |
| -6 | Jump to the ROM monitor (return to BASIC via a warm start, which may also be done by a GO at $0000). |
| -7 | Jump to the routine last loaded by LOAD $. You may also set this up for your own USR function just as in the BASIC manual except that the vector is at $F0 (decimal 240) and a parameter to be passed must appear after the parentheses: USR(-7) 2*B(I)+J. To create a machine language file, see CREATE. |

## DEBUGGING AIDS

HEXDOS adds some powerful program analysis features to BASIC. It retains the simple break on [ctrl C], and will also stop on [repeat] or [break] (see Appendix D). This action may be disabled by POKE 530,1 .

### TRACE

To have a full trace of your BASIC program as it executes, POKE 530,2 before you RUN or CONT it. This will cause BASIC to print the line number in [ ] after each statement is executed. Program output is intermixed with this trace, allowing you to see the action of each line as it executes.

### SINGLE-STEP

The single-step mode uses tracing as above but pauses and waits for a keystroke after each statement. POKE 530,3 enters the single-step mode. After each line, [line feed] will step to the next statement, while [return] will disable the single-step and trace options and continue normal execution. To break out of the program to print variables, list the program, or correct the problem, hold [repeat] or [break] (depending on hardware) and press [line feed]. You may also activate trace or single-step by placing the appropriate POKEs in your program. POKE 530, 0 disables any of these options and returns to normal execution.

## REAL-TIME CLOCK

HEXDOS includes precisely the type of real-time clock supporting software that OSI mentions in their literature on the 610 board. Two jumpers must be added to the 610 board at regular tie points to activate this option as noted in Appendix D.

With the hardware activated, there is a three byte clock which counts seconds since the last cold or warm start. The contents of this clock are available through USR.

Before BASIC tries to execute a line of a program or a line typed from the keyboard, it will do a GOSUB 0 if the contents of memory location 236 and 239 (decimal) are both zero. Location 236 is intended to be an enable/disable switch, while 239 is part of the real-time clock. Location 238 is incremented for each trigger of the NMI line (pulsed once per second to make this act as a clock). Location 239 is the overflow. POKE the two's complement of the desired time delay into 238 (low byte) and 239 and POKE 236,0. If BASIC is running when the time interval ends, it will do a GOSUB 0. Your timeout routine must therefore begin at line 0 and end with a return.

You can also treat this feature as an ON INTERRUPT GOSUB by connecting your interrupting device to NMI and POKEing 255 into both 238 and 239 (and 0 into 236).

One caution which may prevent some frustration with erratic results: the disk motor turnoff routine uses location 238, and every disk track seek resets it to 252 (4 seconds until turning off). This may cut up to 252 seconds off of your preset time interval for each disk access, depending on exactly when the disk accesses occur.

Even though this setup leaves something to be desired in the way of elegance, it does permit some real-time programming. For instance, in a game you can have something happen if the player waits too long to react. A GOSUB cannot be safely executed, however, until the current statement is finished executing. This means that your routine will not be triggered until after the user presses RETURN if the clock runs out while an INPUT statement is being executed, or until he presses a key if you use USR(0).

## PROGRAM CONVERSIONS

BASIC programs saved on tape will be compatible with HEXDOS. Load them just as you normally would, with LOAD.

Programs which use USR should be changed to use USR(-7) by moving the start vector to 240 ($F0) and placing the parameter to be passed immediately after the parentheses rather than within. Values returned by USR are handled normally.

Programs which deal with tape will run as is, except that the old function of SAVE to switch output to tape is replaced by

PRINT #2, [data]

You probably will prefer to change your programs to work with data on disk rather than tape, now that you paid all that money for the disk. See LOAD *, SAVE *, and the section on INPUT and PRINT.

Programs written under another DOS may be loaded most easily by saving them on tape and then loading the tape under HEXDOS. To save a program on tape from OS65D, type

NULL 8:DISK!"IO ,03":LIST

(The space between IO and the comma is essential in this line!) Programs using disk under OS65D will need some minor changes to the input and output sections. OPEN must be changed to LOAD and CLOSE must be changed to SAVE (see LOAD and SAVE).

If you are familiar with the internal format of the other system, you may also transfer the program by using HEXDOS to save it, track by track, from its present location in memory to a file on a HEXDOS disk. HEXDOS automatically corrects the line pointers when loading a program, which should leave you with at least a readable program needing a few changes.

## COPYING FILES

To copy a HEXDOS program from one disk to another, LOAD it from the original disk, put the new disk in the drive, and SAVE [filename]. If you have dual drives, this operation can be done without switching disks by using LOAD! between the LOAD and SAVE commands.

Transferring data files can be a bit trickier. The safest way is to write a short program to read a data file and then write it to the new file. If you're sure of what you're doing, LOAD # and SAVE # can be used to transfer tracks directly.

## AUTOSTART

You can have a BASIC program begin executing automatically when you boot up HEXDOS. If your disk drive is set exactly to specs (300 RPM) the program can be about 170 bytes long. Allowable length decreases as disk speed increases. If this seems short, remember that it can include a command like RUN "BIGFILE", and BIGFILE can be as big as your memory allows, which is about 10K bigger than it could under OS65D.

The size limit is caused by the way the autostart program is stored. On a standard drive, there are about 170 bytes left over on track 0, and this is where the program is kept. The track-writer routine in HEXDOS allows only 2K bytes on all tracks but track 0. On track 0, however, it writes the operating system and then keeps going into the program space until it detects the index hole.

To create or change the autostart program on a disk, boot up HEXDOS and load the program you want to autostart. Please note that the menu program provided on your original HEXDOS disk appears only on track 0, so you should probably do your experimenting on a copy of HEXDOS other than your master. On a non-protected disk, with the desired program loaded, type SAVE#0,768. Reboot from that disk to confirm that your drive will save a program as long as you are attempting to use. If you hold the repeat or break key during the boot sequence, the program will be loaded but not executed.

# Utilities and Demo Programs

## FORMAT

This utility program provides a way to format new or used disks so they will be compatible with HEXDOS. It does so by disabling the normal error traps in the disk handlers and then writing nulls on each track. It also names and dates the disk and sets up an empty directory. Please note that this process *erases* the entire disk.

The only interaction FORMAT requires from the user is to load the disk to be erased and provide the name and date for the disk. HEXDOS is automatically copied to track zero of the new disk so it will boot up with [break] D.Please note that this feature is provided as a convenience for your own use and is not a license to produce copies for others.

## CREATE

This utility program is used to create empty data files. It is not needed for program files, because they are created automatically by SAVE [filename].

The only user input required is the name of the file and its length in tracks (2K bytes per track). Since the length must be specified in advance and may not be changed, it should be set up somewhat longer than you expect to need. Unused space will be filled out with nulls.

To make a machine language file, first use CREATE to make a file *n* tracks long, where *n* is the number of 2048 byte tracks required. Then load the machine language routine into memory (not necessarily at its normal location), and place the proper start address in the two bytes immediately preceding its first byte, low-order byte first. Now use SAVE # to save an image of memory starting at (start address)-2 on consecutive tracks of the file you have created for it. LOAD $ will now load the machine language file to its proper location (with the address in the two bytes immediately before it) and set up the USR(-7) vector to point to the start address of the routine.

## DELETE

This utility program provides a way to delete obsolete files from a disk (HEXDOS itself does not allow you to delete a file once it has been created). DELETE will list the files in the directory one at a time and pause for a command to either retain the file or delete it. To delete the file, press D three times. To retain the file, press any other character, such as the space bar. Do not interrupt this program, since it might leave the directory disagreeing with the actual contents of the disk.

## DISASSEMBLER

This program interactively disassembles the machine code in an area of memory specified by the user. It first requests the address of the desired area. Respond with the address in hex using any appropriate number of digits.

The disassembler will print one screen full of information and then pause. Typing a C (for continue) will resume the listing for another page, while any other character will return to a request for a new start address.

Instructions are printed in standard 6502 mnemonics, but the addressing mode is specified by appending a letter from the table below. The operand is printed in hex. Each byte is also printed in ASCII to the right of a semicolon after the instruction (the high-order bit of each byte is ignored).

| | |
|---|---|
| blank – | absolute, implied, or relative |
| # – | immediate |
| A – | accumulator |
| Z – | zero page |
| X – | absolute,X |
| Y – | absolute,Y |
| S – | zero page, indexed* |
| @ – | indirect |
| X@ – | indexed indirect |
| @Y – | indirect indexed |

* The 6502 recognizes zero page, indexed as being indexed by the X register unless X is involved in the instruction, in which case it uses Y as the index register.

## DEMONSTRATION PROGRAMS

The programs included in this part of the HEXDOS package are useful or entertaining in their own right, but they are also intended to demonstrate some techniques for fully using the power of the HEXDOS system. CHECKBOOK demonstrates the use of serial disk files, requiring that an entire data file be read into memory and then manipulated by the program. ADDRESSBOOK, on the other hand, makes use of random-access files, bringing in only the desired record at any given time. RANFILE provides the necessary subroutines and a simple program to demonstrate usage of random-access files.

SURROUND demonstrates a means for accessing the keyboard in a real-time program which must continue running whether or not a key is pressed. BACKGAMMON shows how to use HEXDOS to reduce requirements for keyboard debouncing, decoding, and status detection in your program. Both SURROUND and BACKGAMMON use the tone generator (see Appendix D) for sound effects.

$LIFE is a machine language program which may be loaded by name. BSR shows another way of handling machine language by using a BASIC program to set up the machine code and link it to USR. This also shows how to use the USR(-7) function in HEXDOS.

## CHECKBOOK

This program uses a serial data file to maintain the ledger for your personal checking account. It records the check number, date, amount, and the payee or other notation used on the check. Deposits, service charges, regular monthly deductions, etc. are best handled by assigning them a unique series of "check numbers." The program is menu-driven, which means that you simply select from a menu of possible actions at any given time. Entering the checks as you write them may seem somewhat tedious at the time, but the real benefit comes with the arrival of your monthly statement. Select "balance statement" from the menu and enter the check and deposit numbers which have cleared the bank during the month. The program provides the balance which should currently appear in your checkbook as well as the balance which should appear on your bank statement.

## ADDRESSBOOK

This program is similarly menu-driven, but it uses random-access files and accesses the disk only when necessary. Record lookup is accomplished using the Soundex code, which allows for slight misspellings in the given last name. Since such a system can find several similar entries, it produces a query after finding any record which is close to the desired last name, requiring a yes or no response. Since there may be several entries with the same last name, even an exact match requires a response.

## SURROUND

This is a real-time video game in which two players attempt to avoid running into barriers on the screen: the border, the opponent's trail, and their own trail. The player who starts on the left side of the screen uses the keys A, W, S, and Z to turn his piece; the player on the right uses L, P, ; (semicolon), and . (period):

```
       ↑                  ↑
       W                  P
  ←A       S→        ←L       ;→
       Z                  .
       ↓                  ↓
```

It is not necessary to press the key hard (this may be hard to remember in the heat of the battle), but you must lead your turns slightly. A quick stab at a key is not very effective; there is no effect of holding a key after the turn is made, so the most effective strategy is to press and hold the key until you see that it has taken effect.

At some time, it will become apparent that the two players can arrive at the same point at exactly the same time and not cause a collision. They may cross trails or even run as one for a time after such an encounter, which adds some interesting twists to the strategy of the game.

## FIFTEEN

In this game, you and the computer alternate picking without replacement from the set of numbers 1 through 9. When either player has collected any three numbers which add to fifteen, he wins.

## REVERSI

This is a board game in which one person plays against the computer. Instructions are given at the beginning of the game. The tone generator is used for sound effects and to warn against illegal moves.

## BACKGAMMON

This is the standard game of backgammon designed as an automated board and dice-roller for two players. The inner table is on the left side of the screen, the bar is in the center, and pieces removed from the inner table at completion are not visible.

When the dice are rolled they are displayed below the table, with the player to move indicated at the right. To move a piece, point to it by pressing the key marked on that point, and then the key on the destination point. The program will ignore illegal moves. You may return a piece to the point from which it came with no penalty until it is placed on another valid point, at which time the move is permanent. More than five pieces may be placed on a point, but they are stacked up so that the screen will never appear to have more than five pieces on a point. This program uses the tone generator for additional feedback to indicate when a move or capture has been accepted.

## $LIFE

This is a simulation of the birth and death of organisms living in a square grid simulated on the screen. Each cell has eight total neighbors, counting the four which share its edges and the four which share its corners. If a cell is touching only one or no living neighbors, it dies in the next generation due to isolation. A cell which has four or more living neighbors dies due to overcrowding. Exactly three live cells touching a given cell will produce a living cell in the next generation.

To run this simulation, first LOAD "$LIFE" to load the machine language file and link it to BASIC. Then use ?USR(-7) to jump to the program. To set up your initial pattern, use A, S, W, and Z as in SURROUND to move the cursor around the screen (it is invisible). To place a live cell, press M; to kill a cell, press the space bar. You may see where the cursor is by pressing M and then the space bar. To start the simulation, press G (go) with either [shift lock] or the left shift key depressed. The simulation will run continuously while the shift lock is depressed, or may be stepped through single generations by releasing the shift lock and pressing a shift key for each step.

## BSR CONTROLLER

This program allows control of lights and appliances using the BSR home control system marketed by Sears and Radio Shack.

For the hardware part of the interface, simply set up the tone generator as suggested in Appendix D, but substitute a 40KHz (approx.) ultrasonic transducer for the speaker. (A suitable transducer is part #MM1002 available for $6 from The MicroMint Inc., 917 Midway, Woodmere, NY 11598.) Place the transducer very close to the front of the control console, just below the LED.

Running the BSR program sets up a machine language program and links it to USR(-7). USR(-7) will access the BSR system until another machine language program is loaded or the system is re-booted. You can now load your program which uses these controls.

To test the system, you may enter commands from the keyboard as T=USR(-7) *n* where *n* is a number from 1 to 22 which selects a key on the console to "press." Numbers 1 through 16 are the numbered keys, and 17 through 22 are the control keys:

| | |
|---|---|
| 17 – | ALL OFF |
| 18 – | ALL LIGHTS ON |
| 19 – | ON |
| 20 – | OFF |
| 21 – | DIM |
| 22 – | BRIGHT |

Note that the keys are only held for a short time with each call; thus the DIM and BRIGHT functions require multiple calls. To activate ALL OFF, as an example, you would type T=USR(-7) 17 . This will work either directly from the keyboard or when executed in a program. You may also use any valid arithmetic expression to designate the key, as long as it results in a value from 1 to 22 (out-of-range values are ignored). Thus, if A is 81 and B is 9, T=USR(-7) A/3-4*(B-3) is a valid way to activate key number 3. Simply have your program or your inputs on the computer keyboard generate the same console actions as you would require to do the job manually.

# Appendix

# Appendix A
# ERROR MESSAGES

Error messages from BASIC or HEXDOS consist of a ? followed by a two letter error code and the word ERROR.

| Error # | Error code | Problem |
|---|---|---|
| 23 | DT | Disk Track (wrong one read) |
| 234 | Fd | File doesn't exist |
| 38 | OR | Out of Room (end of file) |
| 57 | EA | file Exists Already |
| 37 | RO | Read Only (disk is write-protected) |
| 42 | IN | INput file (and you're trying to output to it) |
| 7 | DF | Directory Format (something wrong in it) |
| 104 | PX | Program extended (beyond end of file) |

Error messages generated during input or output will be printed to the device being accessed by that operation. During input from a disk file, this will cause an F ERROR. To see the original error message, temporarily change the INPUT or PRINT statement to use device 0 (keyboard or screen) so that the message will be printed normally.

You can have BASIC jump to a specified line number on any of the above errors, rather than issuing the error message. Your error-handling routine must start at a line number that is a multiple of 256 and less than 32767. POKE the line number divided by 256 into 237 (decimal), and any disk error will cause a GOTO to your routine. In the first line of your routine only, you may find the number of the line where the error occurred by

```
EL=PEEK(135)+256*PEEK(136)
```

If EL is greater than 65280, the error occurred in a line typed from the keyboard. To disable your error routine and restore normal error messages, POKE 237,255. The error handler in HEXDOS does this each time it is called, to prevent your routine from getting into an infinite loop in case of an error in your error-handler.

Your routine can determine what type of error triggered it with USR(-1), which returns the corresponding error number given in the table above. If USR(-1) returns 0, there was no error — you got to where you are via standard BASIC branching.

## Appendix B
## DISKETTE FORMAT

Track zero has its own unique format to comply with the requirements of the bootstrap firmware in ROM on the 600 board. The first byte is the high-order byte of the load address, followed by the low-order byte, and the number of pages of data on track zero.

The remaining tracks start with a sync character ($57) and the track number in binary, followed by 2048 bytes of data. The directory resides on track one as a BASIC program. The line number of the name of each file is the starting track number of that file. It includes all tracks up to but not including the first track of the next file. The last line of the directory contains a *, indicating the first available track.

Note that if you attempt to edit the directory (definitely possible), any BASIC reserved words appearing in filenames will be stored in their compressed form. This means that the filename will appear correct in the directory, but will apparently not exist when you attempt to load it. For instance, CORE would be saved as ASCII C, the token for OR, and ASCII E. If you wish to edit the directory without worrying about hidden reserved words, use lower case. You must then use lower case to load the file, of course.

BASIC programs are stored in core-image form, with the first byte of the program on the first track being loaded to the beginning of BASIC's program space. (Under HEXDOS, this is $0B01. See Appendix C for a complete memory map.)

A machine language file is specified by a filename beginning with $. The first two bytes in a machine language file are the load and start address of the program, with the low-order byte first. These two bytes will be loaded into the two bytes immediately preceding the address they specify, and also into $F0, linking the machine language routine to USR(-7). A machine language file may consist of more than one track just as may a BASIC program, but only the first track need contain the load address (the following tracks are assumed to be contiguous).

## Appendix C
## MEMORY MAP

| Location | Contents |
|---|---|
| 00 | warm start jump |
| 03 | prompt jump |
| 0A | USR jump |
| 0D | NULL |
| 0E | POS |
| 0F | terminal width |
| 13–5A | text buffer |
| 79–7A | start of program |
| 7B–7C | start of variables |
| 7D–7E | start of arrays |
| 7F–80 | end of arrays |
| 81–82 | start of strings |
| 83–84 | end of strings |
| 85–86 | end of memory |
| 87–88 | current line number |
| 8B–8C | start of current stmt |
| 8F–90 | data pointer |
| BC | GETCHR code |
| C2 | REGETCHR code |
| D8 | current track number |
| D9–DD | temporaries |
| DE | start track of program |
| DF | end track of program – 1 |
| E0 | editing flag |
| E1 | head position on inactive drive |
| E2 | I/O device number |
| E3 | I/O mask |
| E4–E5 | used by C1E and C1S |
| E6 | ignore seek error |
| E7 | temporary |
| E8–EA | clock |
| EB | error number |
| EC | nonzero disables ON ERROR GOTO |
| ED | high byte of line # of user error handler |

| Location | Contents |
|---|---|
| EE–EF | time until GOSUB 0 |
| F0–F1 | vector to USR(-7) |
| F2–F3 | temporaries |
| | |
| 0200 | cursor |
| 0201 | character at cursor |
| 0202 | character being output |
| 0212 | debug control |
| 0236–02FF | data file headers |
| 0300–0AFF | HEXDOS |
| 0B00– | BASIC workspace |

# Appendix D
# HARDWARE MODIFICATIONS

None of the following mods is essential to use HEXDOS, but each adds a bit of usefulness to the system. If you are not familiar with procedures for working on integrated circuits, please have someone who is make the solder connections. For temporary use of the mods which just require jumpers between provided tie points, you can simply place wires (#22 or smaller) in the holes without soldering, but you may have some difficulty with intermittent operation using this method.

## REAL-TIME CLOCK

This requires that you add jumpers at tie points provided by OSI to activate the interrupt-driven clock. The two jumpers necessary are added to the 610 board.

The square pad nearest pin 9 of U11 is the source for a pulse every second. There is a row of four square pads across the ends of U10 and U11. These pads connect to the interrupt inputs of U72. The second pad from the end nearest U10 is the interrupt used by HEXDOS. Connect this pad to the one-second pulse line.

Near pin 40 of U72 are 3 square pads arranged in a triangle. Connect the one nearest U72 (U72's interrupt output) to the pad nearest J3 (the NMI line on the bus). The clock will be active the next time HEXDOS is used.

## DISK MOTOR CONTROL

On some models, OSI made an error in the polarity of the MOTOR-ON signal, which they fixed by permanently tying the line to ground. There is a spare inverter on the 610 board which will correct the polarity. First, on the adapter board which connects the disk cable to the 610 board, remove the jumper between pins 4 and 12. If a land has been cut at pin 4, use the old wire to replace it.

Back on the 610 board, cut the land running from pin 14 of U72. Connect pin 14 to pin 3 of U5, and the separated land to pin 4 of U5. You will now have motor control with the real-time clock of HEXDOS, which will deselect the drive and turn off the motor if the disk is not accessed for a short time. The disk is automatically reactivated by any subsequent access.

## BREAK KEY

The break key is wired to the hardware reset line of the computer. It is, alas, in a location where it is often accidentally pressed. You may alleviate this problem and make this single key do the work of [ctrl C] as follows.

Disconnect the break key from the reset line and ground by cutting those lands. (Be careful in cutting the ground land as it must still continue past the disconnected key). You will need to add a normally open pushbutton somewhere out of the way to replace the function of the break key. I suggest just behind the keyboard where it is accessible but not hazardously so.

Now connect the break key to R0 and C7 (the lines to the repeat key), and connect the repeat key to R5 and C2, which will change it to the repeat-last-command function as noted under special keyboard features.

## AUTOMATIC POWER-ON RESET

This mod simply takes advantage of the automatic reset provided to U72 and routes it to the 600 board via an unused pin in the ribbon cable.

On the 610 board, tie pin 34 of U72 to pin 11 of J1. Then, on the 600 board, tie pin 11 of J1 to pin 40 of U8. Now the system will automatically reset and give the D/C/W/M prompt any time it is turned on.

## TONE GENERATOR

This uses the RTS line of the 6850 on the 610 board (the floppy ACIA at $C010, not the cassette ACIA at $F000), toggled rapidly by software, to drive an audio amplifier and speaker. Use an LM386 or equivalent for the amplifier. A convenient place to mount it is between U11 and U67 on the 610 board. Connect the input to pin 5 of U71, and the output to your choice of speaker The BASIC program below will generate a continuous tone of about 800 Hz for testing purposes.

```
10 T=USR(2860)
20 GO TO 10
```

# Appendix E
# RESERVED I/O DEVICES

Device selection is done by a skip chain rather than a table of driver addresses. The only convenient way to implement the additional devices is by intercepting the existing routine.

The input and output routines are called through vectors at $0218 and $021A respectively, as the C1P manual shows. Just change the vectors to point to your drivers (a warm start restores the values used by HEXDOS). If you wish the other devices to remain active, you must test the device number (see memory map) for the number you wish to assign to your device. If the device number is not one of yours, the routine should do a JMP to the address which was originally in the vector. Your routine must preserve all register values and should not tamper with page 0 below $F2 unless you know what you're doing. It can terminate with an RTS or simply jump to the normal I/O routine. (The normal routine accepts 0 or 1 for the keyboard and 2 or 3 for the cassette interface.)

Because of the large number of devices (including all the data files), it was impractical to assign each device a single bit, so there is no provision for simultaneous output to more than one device. One possible way to implement this would be to set up device 1 to set the device pointer to 0, JSR OUTPUT, set the pointer to 2, JSR OUTPUT, reset the pointer to 1, and RTS. Note that you must preserve the registers during this sequence.

The character to be output is passed in the A register, and an input character is expected in the A register. The X register will sometimes have the current POS. It may always be obtained at $0E, and the terminal width at $0F.

I suggest removing the disk or using a blank one during all machine language testing to preclude disasters!

## Appendix F
## STRING MANIPULATION WARNING

HEXDOS does not defeat the infamous "garbage collector" routine and its bugs (see PEEK(65), March 1981 issue). Disk buffers share the string space, so this routine will destroy all file buffers which are open when it is called. The FRE(X) function calls the routine, and it is also called automatically whenever free memory is less than 256 bytes.

If your program does extensive string operations, the garbage collector may interfere with disk operations. There are two ways to get around this:

a) Input your data, do your string manipulations, call FRE(X), and then reopen all data files. This does not work if you must manipulate strings as they are read from the disk.

b) Put the file buffers in a protected memory area. At the beginning of your program, prior to use of any strings, open all data files. After opening all files, include this statement:

```
T=PEEK(129):POKE 133,T:T=PEEK(130):POKE 134,T
```

This decreases the apparent memory size by the amount used by the buffers, but protects them from BASIC. If you run the program repeatedly, the end-of-memory pointer will eventually work its way down until you have no free memory left. Either re-boot HEXDOS from disk or POKE the original values back into 133 and 134.

## Appendix G
## UPDATING OLDER VERSIONS

The major change you will need to make when updating programs written with older versions of HEXDOS concerns filenames. Because filenames are now strings, you will need to go through your programs which open data files and enclose the names in quotes. Also, you must of course enclose filenames in quotes when typing them from the keyboard. Directories created under older versions of HEXDOS may include some files which become unfindable with newer versions. See appendix B for tips on resolving this. Any files created under version 4.0 or later should not have these incompatiblities. If you get a TM ERROR when attempting to do a disk operation, check to see if you forgot the quotes. If so, BASIC thinks it found a numeric variable when it expected a string.

If you used the extra features of GOSUB under version 3.0, you will find that they no longer work. GOSUB [string] is no longer necessary for its original purpose since files can now be opened with strings for filenames. GOSUB [hex value] must now be done with T=USR(-5)[decimal value].

If you used the machine language call of the real-time clock, HEXDOS now does a GOSUB 0 rather than calling machine code. See the section on the real-time clock for correct usage.

## NOTES

## NOTES

## NOTES

The 6502 Program Exchange
2920 West Moana
Reno, NV 89509