PERCOM

MICRODOS™

Disk Operating System
SYSTEMS MANUAL

DATE: JANUARY 1979

PERCOM DATA COMPANY
211 N. KIRBY
GARLAND, TEXAS 75042

PERCOM MICRODOS
VERSION 1
August 24, 1979

Created by
James W. Stutsman

PERCOM DATA COMPANY
211 North Kirby
Garland, Texas 75042
(214) 272-3421

IMPORTANT NOTICE

At several places within this manual references are made to the "TRSDOS & DISK BASIC Reference Manual" published by Radio Shack. This manual is available from Radio Shack (catalog number 26-2104).

WARNING

It is assumed that the reader of this manual is experienced in the use of Radio Shack Level II BASIC (tm). If this is not the case steps should be taken to master Level II before continuing in this manual, as it assumes such knowledge as a prerequisite for using MICRODOS.

A great deal of time and effort has gone into the creation of PERCOM MICRODOS. In spite of this it is still possible that this software may contain some "bugs". If you should discover a problem with MICRODOS, please let us know about it. The best way is to send us a letter describing all of the factors present when the error occured. Be sure to include all information necessary to recreate the problem, as it is very difficult to solve a problem that cannot be recreated. It is best NOT to phone us about a software problem. There is no guarantee that someone familiar with the program will be available when you call, delaying the solution. It is also very difficult to describe software problems over the phone, and even more difficult to solve them that way.

READ THIS FIRST

Disk files created by MICRODOS are NOT compatible with TRSDOS or other disk operating systems. Attempts to read MICRODOS compatible disk files or programs with TRSDOS will fail.

While a utility program exists for MICRODOS to read TRSDOS BASIC program files, the syntax of disk I/O statements in TRSDOS BASIC may not be correct for MICRODOS. It will be necessary to modify such programs to be compatible with the MICRODOS BASIC syntax.

You may also transfer programs from one system to the other by saving the program to cassette while under one DOS then loading the program from cassette while under the other DOS.

MICRODOS is configured for the disk drives sold by Percom Data Co. The Percom disk drives permit operation on 40 tracks with a track to track step time of 20 milliseconds. If you are not using MICRODOS with a Percom disk drive it may be necessary to alter these parameters.

Disk drives sold by Radio Shack are capable of only 35 tracks of storage and will not step faster than 40 milliseconds. If you are using MICRODOS on a system which contains one or more Radio Shack disk drives you must apply the following patches for proper operation.

1. Load MICRODOS and exit to BASIC.

2. To change the number of available tracks to 35, enter the following:

   ```
   POKE 17554,35 : CMD"M",0
   ```

   This will set the track limit to 35 and will write the modified MICRODOS to drive #0.

3. To change the track to track step time to 40 milliseconds, enter the following:

   ```
   POKE 17556,3 : CMD"M",0
   ```

   This changes the step time and writes the modified MICRODOS to drive #0.

I. GETTING STARTED

Buying a new software item is like buying most new things -- there is a tremendous urge to try it out right away before reading instructions or anything else. If you want to use MICRODOS "instantly" by all means do so. The diskette provided contains several sample programs to play with. PLEASE DO NOT REMOVE THE WRITE PROTECT TAB FROM THIS DISKETTE! It is there for your protection to prevent inadvertent destruction of data on the diskette you purchased. Play with MICRODOS, experiment, and enjoy it. When you are ready to learn more about it return to this document. The MICRODOS PRIMER at the end of this manual contains a quick summary of the MICRODOS commands.

II. MICRODOS OVERVIEW

A Disk Operating System (DOS) is, loosly speaking, one or more programs which provide access to a magnetic disk storage device. Typically these programs are large and complex and place many restrictions on the programmer. Data to be stored on disk must be organized in "files" which must be "opened" to be used and "closed" when no longer needed. Such restrictions are a necessity on large computer systems to maintain efficient, orderly usage. Traditionally microcomputer Disk Operating Systems have been designed as replications of big computer systems, taking a heavy toll on the limited resources of the much smaller machine.

The philosophy at Percom is that a DOS for a microcomputer should make minimal demands on available resources while providing essential functions in accessing the disk. These functions include program save and load and a simple means of storing and retrieving data. Random access of the disk should be as easy to perform as sequential access.

MICRODOS was designed around these principles. Its requirements are small -- less than 7K of memory and only 5% of a diskette. Once MICRODOS has been loaded into memory (by power-on or reset) it is no longer dependent on the disk for anything. This, along with the low memory requirement, makes a 16K one-disk system quite useful.

III. DISK ORGANIZATION

Up to four disk drives may be accessed under MICRODOS. These drives are referenced using numbers from 0 through 3. The cable position for drive 0 will be either the connector closest to the Expansion Interface or that furthest away from the Expansion Interface. Which one you have depends on the manufacturer of your cable. On cables made by Percom Data Company position 0 is the furthest from the Expansion Interface.

Data is recorded on a diskette by creating patterns of magnetization in the ferromagnetic material on the surface of the diskette. The read/write head of the disk drive is capable of moving across the surface of the diskette to discrete positions called "tracks". A disk drive may be capable of reaching 35, 40, or even 77 tracks depending on the model used. These tracks may be thought of in the same context as grooves on a phonograph record. Data within a track is stored in segments called "sectors". Under MICRODOS the unit of disk storage is one sector.

Sectors on a disk are accessed by number. The first sector on the first track is number 0, while the number of the last sector on the last track will depend on what model of disk drive is being used. Numbering of sectors on a diskette is similar to the numbering of elements of an array in BASIC.

Accessing a sector on disk, however, requires more than just the sector number. As mentioned previously it is possible to have as many as four disk drives. Therefore a drive number must also be provided for accessing a sector. In accessing a sector under MICRODOS the drive number and the sector number are combined in a single 5-digit number of the form DSSSS. The first digit, D, represents the drive number and may range from 0 to 3. The last four digits, SSSS, are the desired sector number. Range of the sector number is dependent on the type of drive being used. Following are some examples of the DSSSS coding:

```
10010 will access sector 10 of drive 1.
20349 will access sector 349 of drive 2.
00020 will access sector 20 of drive 0.
   20 will also access sector 20 of drive 0.
```

Note that leading zeroes are not required in a DSSSS number. Due to this fact sectors on drive 0 may be accessed using a sector number alone, although there is no harm in also specifying a drive number too. Note also that if a drive number is specified leading zeroes are required on the sector number.

When writing programs to run under MICRODOS the DSSSS number may be specified in several ways. It may be specified as an actual number as shown above, although this is very limiting. MUCH MORE USEFUL is the ABILITY TO SPECIFY 'DSSS' as a VARIABLE or a FUNCTION in a BASIC program. When a variable is used it is first converted to an integer value (if not already an integer). Arithmetic expressions may also be used for DSSSS as long as the result is a positive, legal value.

Some examples of DSSSS expressions are:

DS	a simple variable.
D*10000+S	an expression.
LOC(1)	a function.

However the DSSSS number is specified it must never be negative. The drive number may not be larger than 3 and the sector number must not exceed the largest possible value for the disk in use.

IV. THE MICRODOS OPERATING SYSTEM

Before learning to use MICRODOS it is necessary to learn a little bit about it. MICRODOS is a machine-language program which resides in less than 7K of RAM memory. The purpose of MICRODOS is to provide the BASIC programmer with the necessary tools to use a mini disk drive for data storage and retrieval. Unlike other Disk Operating Systems, MICRODOS exists in conjunction with Level II BASIC, not apart from it. When the computer is first turned on (or the RESET button is pushed) the Level II ROM loads MICRODOS from the disk on drive 0. After performing some initialization MICRODOS turns control over to Level II. It is important to understand that from that point on all communication with MICRODOS is done through the use of BASIC instructions.

MICRODOS is recorded on sectors 0 to 19 of any disk that it is written on (this procedure is described later). Once MICRODOS has been loaded (by power-up or reset) it is not necessary to keep the MICRODOS disk mounted. MICRODOS resides entirely in memory, thus it does not need to have a "system" disk available as with other Disk Operating Systems such as TRSDOS.

V. THE MICRODOS COMMANDS

For some functions it is necessary to issue commands directly to MICRODOS. This is done using the command 'CMD'. The various forms of this command are discussed in the following paragraphs.

CMD"F",D

Before a new diskette can be used by MICRODOS it must have certain control and timing information written on it. The process of writing this data to a new diskette is called "formatting". This MICRODOS command performs the formatting operation on a diskette. 'D' is an expression indicating the drive number of the disk to be formatted, and must be in the range 0 to 3. When 'CMD"F"' is issued the entire diskette is written with the necessary control information. A second pass is then made to verify that every sector can be read. If a read error is encountered during verification the track on which the error occured will be reformatted in an attempt to correct the error. If the sector still cannot be read MICRODOS counts it and

goes on. At the conclusion of the 'CMD"F"' the function LOC(3) will return the number of sectors that could not be verified with zero indicating no failures. Following a 'CMD"F"' all sectors of the disk will be initialized to an "empty" state. This means that reading of any sector of a newly formatted disk will return no data. An important point to note is that THE FORMAT PROCESS IS INHERENTLY DESTRUCTIVE and will overwrite any data present on the disk to be formatted. Users must make absolutely sure that no vital information will be lost before starting 'CMD"F"'.

CMD"M",D

Since MICRODOS must be on a diskette to be loaded into memory, there must be a way to get it from memory back onto a diskette. This is done by issuing 'CMD"M",D' where 'D' is an expression indicating the drive which contains the diskette that MICRODOS is to be written on. The diskette must be formatted. MICRODOS will be written on sectors 0 to 19 of the indicated disk. Prior to issuing this command it should be verified that no important information is on these sectors. Because MICRODOS resides on sectors 0 to 19, prudent programmers should not use any sector below 20 on any disk that might contain MICRODOS.

CMD"I",D

This command is simply a shortcut. It first performs 'CMD"F",D' and then 'CMD"M",D'. As before 'D' is an expression for the drive to be operated on and must be between 0 and 3.

CMD"H",A$

When a power-on or reset is done a "bootstrap" program is loaded which clears the Video Display and loads the MICRODOS software. The loading procedure takes several seconds. It is often desirable to have something displayed on the screen so that the user of the computer has verification that the computer is "doing something". By using 'CMD"H",A$' a message of up to 128 characters may be displayed every time MICRODOS is loaded. 'A$' is a string expression which contains the message to be displayed. If more than one line is necessary the "down-arrow" key is used to end each line. The following example will set up a three-line commercial message to be displayed at every loading of MICRODOS:

```
CMD"H","PERCOM MICRODOS <down arrow>
BEST DOS AVAILABLE FOR THE TRS-80 <down arrow>
BUY YOUR COPY TODAY!" <enter>
```

It is very important to note that this command effects the copy of MICRODOS in memory ONLY. Therefore to make the command work it is necessary to ALWAYS FOLLOW 'CMD"H"' WITH 'CMD"M"'. This is the only way to get the modified DOS on disk.

CMD"K",D

When the computer is powered on or reset it loads MICRODOS and then asks "MEMORY SIZE?". This can create confusion for an inexperienced operator and can lead to errors. Worse, to use the computer the operator must know enough programming to get at least the first program loaded and running. To solve this problem MICRODOS allows up to 128 characters to be "pre-entered" and stored in the DOS for use at start up time. This is done using 'CMD"K",A$' where 'A$' is a string expression containing the characters to be "auto-typed" at power up. Since the "ENTER" key cannot be embedded in a string the "down-arrow" key is used instead. At power-up the "down-arrow" is converted to "ENTER". In the following example the DOS is set up to answer the memory size with 47245, load a menu program, and RUN it:

```
CMD"K","47245 <down arrow>
LOAD 30,R <down arrow> <enter>
```

As with 'CMD"H"' this command affects only MICRODOS as it resides in memory. Therefore it is ALWAYS necessary to use 'CMD"M"' to make the auto-key function work.

VI. PROGRAM MANIPULATION COMMANDS

MICRODOS allows BASIC programs to be saved on diskette and loaded into memory as needed. The commands necessary to do this are described in this section. Note that programs saved by MICRODOS are saved in a "memory-image" form, i.e. the program on disk looks exactly like it did in memory. Therefore it is not possible to read programs from disk as data and print them.

LOAD DSSSS or LOAD DSSSS,R

The 'LOAD' command loads a program beginning at the sector designated by the DSSSS expression. If ',R' follows the command the program is RUN as soon as it is loaded. Should an attempt be made to load a program which is larger than available memory, an 'OUT OF MEMORY' error will occur. In that event no part of the program will be loaded. The 'LOAD' command performs an internal NEW so no part of a program in memory will remain when 'LOAD' is performed. Note that since the string space reserved by CLEAR is not affected by NEW, it may be necessary to CLEAR a smaller string space to 'LOAD' a large program.

SAVE DSSSS

The 'SAVE' command is used to save programs from memory to disk. 'DSSSS' is an expression indicating the drive and first sector to be used in saving the program. It is the responsibility of the user to insure that the sectors written over by the 'SAVE' are unused. No check is made during the save. At the conclusion of the save the command displays the last sector used. This should be noted to prevent overwriting the

program in future disk access.

MERGE DSSSS

The 'MERGE' command works like the LOAD command with one important exception. There is no implied NEW prior to 'MERGE' and any program in memory is NOT cleared out. Lines from the incoming program are merged in with those of the program in memory. In cases where a line from disk is already in memory, the disk line will replace the one in memory. If memory is exhausted before the 'MERGE' is complete an 'OUT OF MEMORY' error will occur and memory will contain as much of the program as would fit.

VII. DISK INPUT/OUTPUT STATEMENTS

Under MICRODOS each disk sector has the capacity for 255 characters (bytes) of data. Data is passed back and forth from the disk using strings, which also have a 255 character capacity. Therefore, for a data item to be recorded on disk, IT MUST FIRST BE CONVERTED INTO STRING FORM. Only two statements are needed to store and retrieve data to and from disk. They allow for simple management of data on disk. In the next section more complex and powerful techniques are discussed.

PUT A$,DSSSS

Data is put on the disk using the 'PUT' statement. The data to be output must be in a string variable ('A$' above) and a 'DSSSS' expression must be given to indicate where on the disk the data is to be put. The following example shows how the word 'HELLO' could be put on sector 50 of the disk in drive 0:

```
100 A$="HELLO"      'LOAD THE DATA INTO A STRING
200 PUT A$,50       'OUTPUT TO DRIVE 0, SECTOR 50
```

Numeric data must be converted to string form before being put on the disk. In this example the number 3.141592653 (the constant Pi) is put on sector 24 of the disk on drive 2:

```
100 PI$=STR$(3.141592653)     'CONVERT NUMBER TO STRING
200 PUT PI$,20024             'PUT TO DRIVE 2, SECTOR 24
```

While experienced programmers may shudder at the lack of efficiency in these methods, their purpose is to show clearly and simply how data may be recorded on disk.

GET A$,DSSSS

To retrieve strings from disk the 'GET' statement is used. Data is read from the sector identified by the 'DSSSS' expression and transfered into string variable 'A$'. The prior contents of 'A$', if any, is lost. In the following example sector 50 of

drive 0 is read and the result displayed. Sample output is also shown.

```
100 GET A$,50              'READ SECTOR 50, DRIVE 0
110 PRINT LEN(A$);A$       'DISPLAY RESULTS
5 HELLO      (Sample output)
```

Note that the LEN function may be used to determine the actual number of characters read.

Numeric data may also be read, but since it was recorded as string data it must be read back as string data. The following example reads the value of the constant Pi from sector 24 of drive 2:

```
100 GET A$,20024       'READ THE STRING
110 PI=VAL(A$)         'CONVERT TO NUMBER
120 PRINT A$;PI        'DISPLAY RESULT

 3.14159 3.14159       (Sample output)
```

The loss of digits from the PUT example is caused by the use of single precision variables.

VIII. FIELD BUFFERS

The techniques previously discussed for data storage and retrieval are simple to use and moderately efficient for storage of text. However, they are very inefficient for storage of numeric information. MICRODOS offers another method which, while more complex, offers much more efficient use of the disk. In this method data is read and written using "Field Buffers" which are special holding areas in memory. MICRODOS provides four Field Buffers which may be defined by the user. Data fields may be defined within these buffers for easier manipulation by the BASIC program. How these buffers are used is best shown with an example. Suppose we want to keep a list of names, addresses, and phone numbers on disk. Each name will be 20 characters or less, each address will be 40 characters or less, and each phone number will be 7 characters. Using the technique previously discussed each entry in the list would require a sector -- one for name, one for address, and one for phone.

```
DEF FIELD #N,20 AS NA$,40 AS AD$,7 AS PH$
```

The 'DEF FIELD' statement is used to name areas within a Field Buffer for easier manipulation by the BASIC program. 'N' is a numeric expression identifying which Field Buffer is to be referenced. In this case three fields are defined. They are 'NA$' (name) as 20 characters, 'AD$' (address) as 40, and 'PH$' (phone) as 7. Suppose that sector 48 of drive 2 contains the following string:

```
"R. Shack            1111 Tandy Trail              5550000"
```

The following program shows how this data is read and accessed

using a Field Buffer:

```
100 GET #1,20048                'READ THE SECTOR
110 DEF FIELD #1,20 AS NA$,40 AS AD$,7 AS PH$
120 PRINT NA$
130 PRINT AD$
140 PRINT PH$

R. Shack                        (result if RUN)
1111 Tandy Trail
5550000
```

Note how the DEF FIELD allowed data to be "broken down" into pieces for simpler processing. In addition three separate data items are stored on a single sector.

While this is a great improvement, there is more that can be done. Notice that out of 255 characters possible on a sector, this example uses only 67 of them, wasting 188 characters! A few calculations show that the 67 character sequence can be put in one sector 3 times, leaving 54 characters unused. How is this done?

First we need to get some terminology out of the way. Whenever we have a group of fields which logically "belong" together we will call that grouping a "record". In the preceding example one "record" consists of a name, an address, and a phone number. What we would like to do is get three records on one sector. There are several ways this can be done. Probably the most obvious is to give each field in each record a different name as N1$, A1$, P1$, N2$, A2$, and so on. This requires a lot of typing and will usually require a longer statement than the 255 characters allowed by Level II. If the fields are processed one at a time we can put a "dummY" record at the front of the field definitions to skip over the unwanted records. In the following program a sector is read and all three records in it are printed:

```
100 GET #1,20048            'READ THE SECTOR
110 FOR I=0 TO 2            'LOOP 3 TIMES
120 DEF FIELD #1,I*67 AS XX$,20 AS NA$,40 AS AD$,7 AS PH$
130 PRINT NA$               'PRINT CURRENT RECORD
140 PRINT AD$
150 PRINT PH$
160 NEXT I
```

Notice that this program looks pretty much like the previous one except for the addition of the FOR--NEXT loop and a new field 'XX$'. The first time through the loop (I=0) the dummy field XX$ will have a lenght of zero. This will position NA$, AD$, and PH$ as the first items in the sector as in the previous example. However, in the second pass through the loop (I=1) XX$ has a length of 67 (one record). The fields NA$, AD$, and PH$ are now offset 67 characters into the sector. We are actually accessing a second record in the same sector.

If all this seems hard to understand, keep in mind that the field variables NA$, AD$, and PH$ are not data items in themselves, but are actually NAMES of data items. As an example of how this works suppose that your newspaper deliverer is very dull-witted and only leaves a newspaper at house number 123. Further suppose that there is only one sign with the numbers 123 on it for use by the entire neighborhood. The only way for everyone to get a newspaper is to move this number from house to house as each paper is delivered. Accessing data in a multi-record Field Buffer works the same way. The variable names ("house numbers") are moved from record to record ("house to house") so the proper data is retrieved.

But suppose we wish to access all the records of a sector at once. How can we do this? The best way is by using arrays. By mapping each element of an array over a record in the Field Buffer we can have access to all records at once. Using our name, address, and phone number example we could do it this way:

```
100 DIM NA$(2),AD$(2),PH$(2)    'DEFINE ARRAYS
110 GET #1,20048                'READ THE SECTOR
120 FOR I=0 TO 2                'START LOOP
130 DEF FIELD #1,I*67 AS XX$,20 AS NA$(I),40 AS AD$(I),7  AS
PH$(I)
140 NEXT I
200 FOR J=0 TO 2                'PRINT LOOP
210 PRINT NA$(J)
220 PRINT AD$(J)
230 PRINT PH$(J)
240 NEXT J
```

At first glance you may be tempted to say that this won't work. It does look a lot like the previous example, but the array subscripts are the big difference. Because of subscripting NA$(0) is NOT the same name as NA$(1). So what we are doing is giving each record in the sector a unique name rather than moving the same names from record to record.

Thus far we have had a great deal of discussion about how to read data into a Field Buffer and access it, but nothing at all about how the data gets on the disk in the first place. Before data can be put into a Field Buffer the fields need to be defined as they were before using the 'DEF FIELD' statement. Getting data into these defined fields is where things get tricky. Recall that we said earlier that the variables created by the 'DEF FIELD' statement were NAMES of data items rather than data items themselves. Suppose we try the following example program:

```
100 DEF FIELD #1,10 AS A$
110 A$="HELLO"
120 DEF FIELD #1,10 AS B$
130 PRINT B$
```

When we print 'B$' we expect to get a friendly "HELLO" greeting. Suprise! We get nothing of the sort! (What we get depends on whatever garbage was in Field Buffer 1 before we started.) What happened? Examining the program, line 100 seems alright. We

named the first 10 characters of Field Buffer #1 as 'A$'. The "gotcha" comes in line 110. It appears that we are moving "HELLO" into variable 'A$', but in reality what we are doing is moving the NAME 'A$' to the data item "HELLO". Thus we can print A$ and get "HELLO", but we can't seem to find it in the Field Buffer. Yes, this is confusing, but it is done this way to make the computer more efficient. So how do we get data into the miserable Field Buffer?

LSET A$="HELLO" or RSET A$="HELLO"

Enter the heroes! The statements 'LSET' and 'RSET' are used to MOVE data from one variable to another. Data is left-justified or right-justified depending on which form is used. If we had used 'LSET' in line 110 of our previous example program, the result would have been that the first 10 characters of Field Buffer #1 would contain "HELLO ". Notice the five trailing spaces. These are provided automatically by 'LSET'. Use of 'RSET' would have resulted in " HELLO". If, however, we had defined 'A$' as having only 4 characters, both 'LSET' and 'RSET' would have resulted in 'A$' containing a four-letter word describing a place that is warm year-round and I don't mean Florida!

Now that we know how to get data INTO a Field Buffer we will look at an example program to produce the data that we read in our previous examples, the name, address, and phone number records.

```
100 FOR I=0 TO 2                    'START LOOP
110 DEF FIELD #1,I*67 AS XX$,20 AS NA$,40 AS AD$,7 AS PH$
120 INPUT "ENTER DATA";N$,A$,P$ 'GET DATA
130 LSET NA$=N$                     'PUT IN FIELD BUFFER
140 LSET AD$=A$
150 LSET PH$=P$
160 NEXT I                          'CONTINUE LOOP
200 PUT #1,20048
```

The last line, 200, actually records the data on disk.

In previous discussions of disk I/O using string variables, the amount of data written to disk was always equal to the length of the string variable from which the data was written. Likewise, the string variable into which the data was read from the disk, is made the same length as the incoming data. When using Field Buffers MICRODOS must have a similar method to determine how many characters to actually write. The rule that is used is that the length of the Field Buffer is assumed to be as long as the total length of fields defined by the longest DEF FIELD statement. If necessary a dummy DEF FIELD can be executed to establish the proper length for the Field Buffer. Following a GET to a Field Buffer the length of the Field Buffer is set to the length of the data read. However, the length may be changed by subsequent DEF FIELD statements. For this reason, it is best to do a DEF FIELD after a GET and before a PUT so the proper length will be written to disk.

While this scheme may seem restrictive, it does insure that no "garbage" data is transferred to or from the disk. The programmer may determine the current length of any Field Buffer using the function 'LOF'. The format of the function call is 'LOF(N)' where 'N' is an expression indicating which Field Buffer's length is to be returned. Note that '#' is not required preceding the Field Buffer number expression. Programmers familiar with the use of 'FIELD' in Radio Shack Disk BASIC should pay particular attention to the concept of length in conjunction with Field Buffers. Radio Shack Disk BASIC assumes a length of 255 bytes and always transfers that many characters with each GET or PUT. MICRODOS users desiring this feature should always do a dummy DEF FIELD for 255 characters immediately before each PUT.

Our discussions involving I/O using Field Buffers have all been oriented toward strings up to this point. How does one handle numeric data efficiently? Since only string fields can be defined in the DEF FIELD statement, numeric data must be converted to string form. Earlier we did this with the 'STR$' function. This results in needlessly long strings, however, there is a better way.

MKI$(I), MKS$(S), and MKD$(D)

The 'MKX' functions make numbers into strings. These strings, however, contain the numbers in their internal form and are not directly printable. 'MKI$' makes its integer argument ('I') into a string two characters long. 'MKS$' makes a 4 character string out of its parameter 'S', and 'MKD$' builds an 8 character string from the parameter 'D'. Let us refer to an earlier example of storing the mathematical constant PI on disk:

```
100 PI=3.14159                  'DEFINE CONSTANT
110 DEF FIELD #1,4 AS PI$       'DEFINE FIELD
120 LSET PI$=MKS$(PI)           'CONVERT TO STRING
130 PUT #1,10045                'WRITE TO DISK
```

In using the 'MKX' functions one must be very careful to define strings of the proper length for the type of number being converted. Refer to the following table:

NUMBER TYPE	REQUIRED STRING LENGTH
Integer	2 characters
Single prec.	4 characters
Double prec.	8 characters

These function solve only half of the problem. There still must be a way to get strings back into numeric form.

CVI(I$), CVS(S$), and CVD(D$)

The 'CVX' functions perform the string to number conversion. 'CVI' converts string parameter 'I$' to an integer. 'CVS' converts 'S$' to a single precision floating point number and 'CVD' converts 'D$' to double precision. Note that these functions CANNOT be used to change the type of a number, i.e. do not use 'CVI' on a string that was created using 'MKS$'. As a partial protection against this, a 'Function call' error will occur if the string parameter for any 'CVX' function is not the proper length for its number type.

IX. MICRODOS FUNCTIONS

Several of the functions available in MICRODOS have been discussed in previous sections. In this section all other functions that MICRODOS provides are explained.

LOC(N)

The 'LOC' function is used to get various bits of information from MICRODOS. 'N', the parameter, indicates what specific information is desired. If 'N' = 0, MICRODOS returns a DSSSS number for the last drive and sector accessed. This call is useful in keeping track of where a program writes its last data. 'LOC' with a parameter of 1 returns the largest possible sector that can legally be accessed under the current version of MICRODOS. This number will vary depending on the model of disk drive your MICRODOS was designed for. Advance programmers will find the most use for 'LOC' with a parameter of 2. This call returns the status (in decimal) of the 1771 Disk Controller following any disk I/O which results in an error. It is up to the user to analyze the status to determine the problem. A call to 'LOC' using an argument of 3 return, as previously discussed, the number of sectors which could not be verified during a format using 'CMD"F"' or 'CMD"I"'.

&H - Hexadecimal Constants

For convenience to the programmer knowledgable in machine language, any constant preceded by '&H' will be interpreted as a hexadecimal constant. The number may be from 1 to 5 hexadecimal digits. Constants of this type always represent signed integers. Hex constants may not be used as response to an INPUT statement or in a DATA statement. Note that octal (&O) constants are NOT supported by MICRODOS.

INSTR(N,S1$,S2$) - String search function

Searching a string for the first occurrence of a substring may be rapidly accomplished using the 'INSTR' function. String variable or expression 'S1$' is searched for the first occurrence of the substring specified by variable or expression 'S2$'. The

first parameter, 'N', is optional. If present it indicates at what position within 'S1$' the search begins. A value of 1 is assumed if 'N' is not present. The value returned by 'INSTR' is the position in 'S1$' where 'S2$' was found, or zero if no match could be found. In the following examples string 'A$' contains the value "ABCDEFG":

Function call	Result
INSTR(A$,"BCD")	2
INSTR(A$,"XY")	0
INSTR(A$,"ABCDEFGHI")	0
INSTR(3,"ABC")	0
INSTR(3,"123123123","12")	4

Additional information may be obtained from the "TRSDOS & DISK BASIC Reference Manual".

MID$(S1$,N1,N2)=S2$ - String replacement function

Under MICRODOS it is permissible for 'MID$' to appear on the left side of an '=' sign. This may be used to replace a part of a string with a specified substring. 'S1$' is the string variable to be operated on. 'N1' specifies the starting position for replacement to begin. 'N2', which is optional, specifies how many characters are to be replaced. 'S2$' is a variable or expression for the replacement string. If 'N2' is omitted either LEN(S2$) or LEN(S1$)-N1+1 will be used, whichever is shorter. In the examples below string 'A$' contains "ABCDEFG":

Expression	Resulting A$
MID$(A$,3,4)="12345"	AB1234G
MID$(A$,1,2)=""	ABCDEFG
MID$(A$,5)="12345"	ABCD123
MID$(A$,5)="01"	ABCD01G
MID$(A$,1,3)="***"	***DEFG

Additional information and examples may be obtained from the "TRSDOS & DISK BASIC Reference Manual".

DEF USRn / USRn - User machine-language functions

Up to ten external user subroutines may be defined ranging from 'USR0' to 'USR9'. The address of the entry point to a user subroutine is specified using 'DEFUSRn=X' where 'n' is the number of the user routine (0-9) and 'X' is an expression indicating the entry point address. The expression 'X' must be an integer in the range -32768 to +32767. User external subroutines are normally used only by advanced programmers familiar with machine language. For additional information and examples refer to the "TRSDOS & DISK BASIC Reference Manual".

DEF FNx - User-defined functions

In addition to the functions intrinsic in Level II BASIC such as LOG, SIN, etc., as LOG, SIN, etc., it is possible for the user to define his own implicit functions. This is done using the 'DEF FNx' statement where 'x' is a legal variable name by which the function will be referenced. For example to define the mathematical constant Pi as an implicit function:

```
100 DEF FNPI=3.141592653
```

Note that this function is single precision, thus some of the digits of the fraction will be lost. Once the 'DEF' has been executed, the user may refer to the function by name:

```
200 A=FNPI*R*R
```

User-defined functions may also have parameters. For example:

```
100 DEF FNAC(R)=3.14159*R*R
200 INPUT "RADIUS";X
210 PRINT "THE AREA IS";FNAC(X)
```

In this case a single parameter, 'R', is defined. Note that should there already be a variable whith the name 'R' it will not be effected by the function definition. Regardless of how many parameters may be defined in a function, the number of parameters in the call must exactly match the number of parameters in the definition. See the "TRSDOS & DISK BASIC Reference Manual" for additional information and examples.

X. MICRODOS STATEMENTS

One new statement is provided for by MICRODOS, an enhancement of the 'INPUT' statement.

LINE INPUT "PROMPT";A$

The 'LINE INPUT' functions much like the normal 'INPUT' statement with the following exceptions:

1. When the statement is executed the prompt, if any, is printed, but no question mark is displayed.
2. Each 'LINE INPUT' may specify only one string variable.
3. Commas, quotes, and colons are accepted as part of the string input.
4. Leading blanks are also accepted.

For additional information and examples refer to the "TRSDOS & DISK BASIC Reference Manual".

XI. MICRODOS ERRORS

All errors are reported by MICRODOS using descriptive error messages. Error code numbers may be derived in 'ON ERROR' routines by using the expression 'ERR/2+1'. Descriptions of error codes 1 through 23 may be found in the "Level II BASIC Reference Manual" published by Radio Shack. In this section all error codes and messages are listed. An explanation of the MICRODOS-related$errors (24 and above) follows. Note that MICRODOS errors 24 and above cannot be simulated via the 'ERROR' statement. An attempt to do so will result in the 'UNPRINTABLE ERROR'.

Error code	Error message
1	NEXT WITHOUT FOR
2	SYNTAX ERROR
3	RETURN WITHOUT GOSUB
4	OUT OF DATA
5	ILLEGAL FUNCTION CALL
6	OVERFLOW
7	OUT OF MEMORY
8	UNDEFINED LINE
9	SUBSCRIPT OUT OF RANGE
10	REDIMENSIONED ARRAY
11	DIVISION BY ZERO
12	ILLEGAL DIRECT
13	TYPE MISMATCH
14	OUT OF STRING SPACE
15	STRING TOO LONG
16	STRING FORMULA TOO COMPLEX
17	CAN'T CONTINUE
18	NO RESUME
19	RESUME WITHOUT ERROR
20	UNPRINTABLE ERROR
21	MISSING OPERAND
22	BAD FILE DATA
23	FEATURE NOT IMPLEMENTED
24	INVALID DRIVE NUMBER
25	INVALID SECTOR NUMBER
26	DISK READ/WRITE ERROR
27	DISK WRITE PROTECTED
28	DISK OVERFLOW
29	DISK MISSING OR DOOR OPEN
30	FIELD OVERFLOW
31	FUNCTION DEFINITION ERROR
32	INVALID FIELD BUFFER NUMBER
33	DISK DRIVE NOT AVAILABLE
34	DISK SEEK ERROR
35	CAN'T FORMAT DISK

Description of errors 24 and above

Error 24 - INVALID DRIVE NUMBER

This error is issued when a drive number is encountered outside the range of 0 through 3.

Error 25 - INVALID SECTOR NUMBER

If the sector referenced in a DSSSS expression is either negative or beyond the limit of the disk drive for which the current version of MICRODOS was designed, this error will be issued.

Error 26 - DISK READ/WRITE ERROR

When this error is issued it indicates that, despite repeated retries, the disk I/O operation initiated in the failing line could not be successfully completed. To get the exact disk controller status use the function 'LOC(2)'.

Error 27 - DISK WRITE PROTECTED

A disk function has been requested which requires writing on the diskette, but the referenced diskette has a write protect tab on it.

Error 28 - DISK OVERFLOW

During execution of a 'SAVE' command the end of the disk was encountered before the last line of the program was written to disk. The program is NOT successfully saved and an attempt to 'LOAD' it will also result in this error.

Error 29 - DISK MISSING OR DOOR OPEN

A legal drive and sector number have been specified, but either there is no diskette in that drive or the door was not closed. This error might also occur if the drive motor were not turning for some reason.

Error 30 - FIELD OVERFLOW

In a DEF FIELD statement either a single field was defined for more than 255 characters, or the total of all defined fields exceeds 255.

Error 31 - FUNCTION DEFINITION ERROR

In processing a DEF FNx statement, the function was not correctly defined, or an FNx reference was made to a function that was not defined.

Error 32 - INVALID FIELD BUFFER NUMBER

A Field Buffer number was either less than 1 or greater than 4.

Error 33 - DISK DRIVE NOT AVAILABLE

The drive referenced in the failing line either does not exist or fails to properly select when accessed.

Error 34 - DISK SEEK ERROR

Following a seek operation (moving the head to a specific track) the disk controller was unable to verify that the head was on the proper track.

Error 35 - CAN'T FORMAT DISK

This error can occur two ways. The first is due to a failure of the write circuitry on the indicated disk drive. The second is more than 255 sectors which cannot be verified.

XII. OTHER VERSIONS OF MICRODOS

It is conceivable that MICRODOS will evolve into one or more new versions. Readers who purchased MICRODOS prior to the completion of this manual may note the absence of any text concerning the 'FIELD LOAD' and 'FIELD SAVE' statements. These statements will be dropped in future version of MICRODOS since they are easily duplicated using LSET and RSET in conjunction with DEF FIELD.

Percom Data Company provides ongoing support for MICRODOS. As bugs are reported and fixed, correctional patches will be made available to registered MICRODOS owners. A patching facility has been built into the MICRODOS diskette which will allow simple correction of bugs.

MICRODOS was originally designed as a vehicle for distribution of application software in a controlled execution environment. Users with unique needs in a DOS are invited to write Percom Data Company for a quotation on custom versions of MICRODOS which can be licensed for distribution of software products.

MICRODOS SYSTEM DISKETTE PRIMER

INTRODUCTION

The MICRODOS system diskette contains four BASIC programs in addition to the MICRODOS operating system. The first of these is simply a menu from which to select one of the other three. It is loaded automatically when MICRODOS is activated by power-up or reset. Control may be returned to this menu program from any of the other programs discussed below by using the command 'EXIT'.

In the following sections each of these programs will be briefly discussed. A section is also included with information on how to get started with MICRODOS.

PROGRAM 1 - A simple Disk File Manager

Disk file management is one of the features usually found in large, complex disk operating systems. Since MICRODOS is not file oriented, it does not need to have such a system "built-in". This program is an example of how a file management system might be developed in BASIC using the disk handling commands in MICRODOS. It maintains a directory of up to 50 files. Each directory contains the file name (up to 10 characters), the starting and ending sector numbers, the file type (P for program, D for data file), and a remark field of up to thirty characters.

All commands in the file manager, with the exception of the EXIT command, may be preceded with a single digit disk drive number (0-3) and a slash (/) to indicate which disk drive the command is to be executed on. If the drive number is not provided, drive 0 is assumed.

The INT command is used to INiTialize the disk directory on the indicated drive. This means writing an empty directory to the specified disk. Obviously any directory information that may have existed on the disk prior to the execution of this command will be lost.

One of the more frequently used commands is the DIR command which is used to display the DIRectory on the screen. It assumes that the specified drive contains a properly built disk directory. All information in the directory is displayed.

To add a file to the disk directory the NEW command is used. It prompts the user for the necessary file information from the keyboard. A directory entry is then constructed and inserted in the directory. If the file is already in the directory the user will be allowed to replace it if desired.

The DEL command allows a directory entry for a file to be DELeted. If the file does not exist, any attempt to delete it is reported as an error.

File information may be updated with the CHG command. It requests the name of the file to be changed and prompts the user through the various entries to be changed. For each item requested, a response of 'ENTER' will cause that item to be left unchanged.

If the command entered fails to match one of those previously described, the directory is searched for a file matching the command. Should such a file be found, it is checked for a file type of "P" (Program). If both conditions are met the program is loaded from disk and executed.

PROGRAM 2 - Disk Utilities

Using the features of MICRODOS most of the classical disk utilities can be written in only a few BASIC statements. This program is included to provide examples of how such utilities might be written. Each utility is called by name and EXIT is used to return to the system menu program.

FORMAT is probably the most essential utility. It is used to write the necessary control and timing information on new diskettes so that they may be used by the computer. The essence of this utility is handled by a single command in MICRODOS. The remaining code is for parameter entry and validation.

Another popular command is BACKUP. It is used to copy one entire diskette to another. This function is essential in business processing to protect against loss of valuable data. This utility formats the destination diskette and copies the source disk to it. If the source and destination disk drives are identical the utility will use all of available memory for disk buffers to reduce copy time. In keeping with the MICRODOS philosophy it is never necessary to have more than one floppy disk drive to use this utility.

If only a part of a diskette is to be copied, the COPY utility is used. It can be used to copy data from one disk to another or even to copy data from one part of a disk to another part of the same disk. All available memory is used for buffers to reduce the copy time. Like all of the MICRODOS utilities the COPY routine uses no machine language routines, only BASIC.

Available space on a diskette can be located using the FREE utility. This routine searches a diskette for empty sectors, an empty sector being determined by having a data length of zero. The number of each empty sector found is displayed on the screen.

The readability of a disk may be checked with the VERIFY command. This command reads every sector on the specified diskette. All sectors which cannot be read are displayed on the screen. A

summary is given of how many bad sectors were found and what percentage of the disk that they represent.

Disk space may be released by the ERASE utility. It writes empty sectors over the range specified, thereby allowing those sectors to be displayed during operation of FREE.

DUMP is a very useful utility for determining what is actually recorded on a file. It reads the sector(s) specified and displays them on the screen as printable characters. Any character which is not in the printable ASCII set is replaced with a graphic "block" (all elements on).

PROGRAM 3 - The Percom 5 1/4 Inch Notebook

This program is a sample application written using MICRODOS. It functions like a notebook in that entries can be written on disk with it and read back. A very simple Table of Contents is maintained for ease in retrieval of information. The notebook comes "pre-loaded" with information about each of the MICRODOS Disk BASIC statements. User data may be added to the unused pages.

Data in the Percom Notebook is structured as "pages" of data. A page is 15 lines of a screen display. Pages are numbered, as in a normal notebook, and are accessed by number. The table of contents is kept on page 0 and is accessed using that page number. Entries in the table of contents consist of a key word of eight characters or less and the page number associated with that key word.

Data is retrieved using the READ command followed by the page number to be accessed. The indicated page is retrieved and displayed on the screen. A flashing prompt then waits for a single key entry indicating what should be done next. Entry of a right arrow will cause the "next" page (i.e. current page + 1) to be displayed. Using the left arrow will cause the "previous" (current page - 1) page to be displayed. To terminate the READ command and return to the command menu, type 'X'.

Data is entered into the notebook using the WRITE command followed by the page number on which the data is to be written. Up to 15 lines of data may be typed on any page. Each line may contain up to 62 characters. The system will prompt for each line by displaying the line number at the left margin of the screen. If the line typed is too long, an error will be indicated and the line will have to be retyped. Should the user desire to enter fewer than 15 lines he may enter a line consisting of only the 'ENTER' key. The page will then be immediately written to the disk. If a blank line is to be written it should be entered as a single space followed by 'ENTER'. After the page is entered and written to disk the flashing prompt will appear. The options

are left arrow, right arrow, and 'X' as described for READ.

Use of the WRITE command to page 0 will cause the program to behave somewhat differently. In this mode the Table of Contents may be updated. A request will be made for a keyword of up to 8 characters. The Table of Contents will be searched for this word. If not found the user will then be given an opportunity to enter it into the notebook. Should he decide to do this, a prompt will be made for the page which the keyword is to reference. Upon locating a keyword in the Table of Contents, the user will be asked if he wishes to update the entry or delete. A 'Y' response will cause a prompt for a page number. Entry of a non-zero page number will cause the old number to be replaced, while a response of 0 causes the entry to be deleted.

Use of the NEW command will generate a completely blank notebook. The drive and sectors to be used are encoded in the first few lines of the program. This version is set to use sectors 200 through 399 on drive 0. It is recommended that this command not be used until the software diskette has been duplicated using the BACKUP utility.

While a brief description of each command is given in the command menu, additional information may be obtained using the HELP command. This command displays somewhat more detailed data relative to the use of the commands. It then returns to the command menu.

PROGRAM 4 - A Disk Diagnostic Test

If a question should arise concerning the operational status of a disk drive, this utility may be used. It is designed to exercise all the functions of the disk drive in order to establish that it is working correctly. This test is performed in three phases.

During Phase 1 the entire diskette under test is formatted. This is a process which writes address and control information on each track of the diskette to make it accessible to the computer. The format process is inherently destructive and WILL DESTROY ALL DATA ON THE DISK. Therefore, before running the Disk Test, BE SURE THE DISKETTE THAT IS IN THE DRIVE BEING TESTED HAS NO CRITICAL DATA ON IT. Do not run Disk Test on the MICRODOS diskette, as it will destroy the data on the diskette.

At the conclusion of Phase 1 a message is printed giving how many sectors, if any, could not be verified after formatting. Phase 2 then begins. During this phase 100 random sectors are written with a special data pattern.

In Phase 2 the 100 sectors are read and checked against the pattern. If any sector fails to compare or if any error is encountered on the disk, a message is displayed indicating the

type and sector location of the error.

At the conclusion a summary is given showing the number of errors encountered during the test.

PROGRAM 5 - Exit to BASIC

This program is actually not a program at all, but a simple means of getting back to BASIC for user programming. When this option is selected a NEW will be performed, clearing the screen and displaying 'READY'. Control may be returned to the MICRODOS System Diskette either by pushing the 'RESET' button or by typing 'LOAD 30,R'.

GETTING STARTED

Before attempting to do anything with MICRODOS it is a good idea to make a backup copy. This may be done by selecting Program 2 from the System Diskette Menu, the Disk Utilities. From the Utility Menu select the 'BACKUP' utility. This will then prompt for the drive number of the source diskette (drive to copy 'from'). Type 0 and 'ENTER'. A prompt will then be displayed for the drive number of the destination diskette (drive to copy 'to'). If you have more than one disk drive, insert a blank diskette in drive 1, type 1, and push 'ENTER'. The backup will then occur automatically.

Single drive users should type 0 and 'ENTER'. A prompt will then display to load the source diskette and push 'ENTER'. When this has been done as many sectors will be loaded as can be put in memory. Then a prompt will display to load the destination diskette and push 'ENTER'. At this time replace the MICRODOS diskette with a blank and push 'ENTER'. The blank diskette will be formatted and the sectors in memory will be written to it. Then a prompt will again display for the source diskette. Replace the MICRODOS diskette in drive 0 and push 'ENTER'. This process will repeat until the entire diskette has been copied.

MICRODOS
A DISK OPERATING SYSTEM FOR THE TRS-80

MICRODOS is a very simple yet powerful disk operating system for the Radio Shack TRS-80 computer. It is completely resident in less than 7K of RAM and requires no disk files to be mounted once the operating system is loaded. MICRODOS operates entirely within the realm of Radio Shack Level II BASIC providing both DOS and Disk BASIC functions. Since all commands are issued in BASIC, MICRODOS does not access the disk through traditional file structures. It offers instead commands to access data anywhere on the disk, thus allowing the programmer full control over the organization and access of the disk.

MICRODOS occupies the first 20 sectors of a disk (sectors 0 -19). No other disk space is used and the MICRODOS diskette may be removed once MICRODOS has been loaded.

MICRODOS is NOT compatible with the Radio Shack Disk Operating System (TRSDOS). Furthermore it may be necessary to modify some programs written in Radio Shack Disk BASIC to function properly with MICRODOS, particularly if the programs create or use disk data files.

MICRODOS COMMANDS:

CMD "F",D

Formats the disk on drive D (0<D<3). Formats the entire disketted with full verification.

CMD "H",A$

Moves string A$ to a special buffer which is displayed when 'Booting'. The String length is limited to 128 characters. Use the 'down arrow' (|) for the NEW LINE character. You must use CMD "M" (described later) to get this heading onto the disk.

CMD "K",A$

Moves string A$ to a buffer which is fed to BASIC as keyboard entries on 'Booting'. Same restrictions as CMD "H". Useful for automatically starting a program.

CMD "I",D

Formats a diskette in drive D (D = 0,1,2,3) and writes a copy of MICRODOS on the diskette beginning at sector 0.

CMD "M",D

Writes the MICRODOS system to the diskette in drive D beginning at sector 0.

BASIC COMMANDS

LOAD DSSSS(,R)

loads the program on drive D, sector SSSS. DSSSS is a number or variable in which the first digit is the drive number (0 - 3), and the last four digits are the sector number. This command may optionally be followed by a comma and 'R' to cause the program to automatically run after it is loaded.

SAVE DSSSS

Saves the program in memory on drive D beginning at sector SSSS. Use the LOC(0) command (described later) to determine the last sector used.

MERGE DSSSS

Merges the program on drive D beginning at sector SSSS with the program already in memory.

DISK I/O STATEMENTS:

GET A$,DSSSS

The statement reads the contents of sector SSSS on drive D into string A$. The length of A$ reflects the amount of data read and may be 0. DSSSS may be a number or a variable.

PUT A$,DSSSS

This statement writes the contents of string A$ to sector SSSS on drive D. A$ may range in length from 0 to 255 bytes (characters). DSSSS may be a number or a variable.

DEF FIELD #N,N1 AS F1$,...,NI AS FI$

Defines a disk buffer N as fields F1$ to FI$, having lengths N1 to NI respectively. Lengths may be expressed as a number or as an expression. If an expression is used, it must be enclosed in parentheses (). Maximum buffer size is 255 characters. Data may be placed in field buffers using LSET or RSET statements in conjunction with MKI$, MKS$, AND MKD$ for number to string conversion. Functions CVI, CVS, and CVD are use to extract numeric data from strings (string to number conversion).

DISK I/O FUNCTIONS:

LOC (N)

This function returns a value depending on argument N.

```
LOC (0) = The last DSSSS accessed
LOC (1) = The largest possible SSSS for the
          current version of MICRODOS
LOC (2) = The disk controller status after
          an error
```

LOF (N)

This function returns a value which is the total length of all fields in buffer N. The value returned will range from 0 to 255.

CONVERSION AND PLACEMENT FUNCTIONS:

```
LSET A$ = B$
RSET A$ = B$
```

These statements are used to place string data in a disk buffer. LSET left justifies within a field, blank filling on the right. RSET right justifies with blank filling on the left.

```
MKI$ (I)
MKS$ (S)
MKD$ (D)
```

These functions are used to store numeric data compactly in a string buffer. They convert integers, single precision, and double precision numbers to strings of 2, 4, or 8 bytes respectively.

```
CVI (I$)
CVS (S$)
CVD (D$)
```

These functions are the converse of MKI$, MKS$, MKD$. They convert numeric string data in string fields back to numeric values.

OTHER FUNCTIONS AND FEATURES:

```
&H - Hexadecimal Contants
INSTR (N,S1$,S2$) - String search function
MID$ (S1$,N1,N2) = S2$ - String replacement function
DEF FNx - User definged functions
DEF USRn / USRn - User machine language functions
LINE INPUT "PROMPT";A$
```

PERCOM DATA COMPANY JUNE 4, 1979

ADDENDUM FOR MICRODOS 1.10 (RELEASED 6-4-79)

WHEN THE MICRODOS SYSTEM DISKETTE IS LOADED (BY POWER-ON OR RESET) IT AUTOMATICALLY LOADS A MENU PROGRAM TO EXECUTE THE PREPROGRAMMED UTILITIES. TO ESCAPE THIS MENU AND RETURN TO BASIC, SIMPLY PUSH THE 'BREAK' KEY.

VERSION 1.10 OF MICRODOS CHANGES SOMEWHAT THE WAY IN WHICH DISKETTES ARE FORMATTED. VERSION 1.00 WOULD ABORT THE FORMAT IF ANY SECTOR WAS FOUND WHICH COULD NOT BE READ AFTER 9 ATTEMPTS TO FORMAT IT. IN REVISION 1 (CHANGING VERSION 1.00 TO 1.10) THE FORMAT OPERATION WAS CHANGED TO ALWAYS COMPLETE WITHOUT ERROR UNLESS THE DISK COULD PHYSICALLY NOT BE FORMATTED DUE TO DISK WRITE BEING INOPERATIVE OR MORE THAN 255 BAD SECTORS BEING FOUND. AT THE END OF THE FORMAT THE NUMBER OF BAD (I.E. UNREADABLE AFTER 9 TRIES) SECTORS MAY BE ACCESSED USING THE FUNCTION CALL 'LOC(3)'. THIS VALUE WILL BE INITIALLY SET TO 0 AND WILL THEREAFTER CONTAIN THE NUMBER OF BAD SECTORS FOUND DURING THE MOST RECENT 'CMD"F"' OR 'CMD "I"' COMMAND.

MICRODOS 1.10 IS BEING SHIPPED CONFIGURED FOR EITHER SIEMENS (*) OR PERTEC DISK DRIVES SOLD BY PERCOM DATA COMPANY. THIS CONFIGURATION ALLOWS FOR 40 TRACKS AND A TRACK TO TRACK STEPPING TIME OF 20 MILLISECONDS. SINCE YOU MAY NOT BE USING MICRODOS ON A PERCOM DISK DRIVE, THE FOLLOWING INFORMATION IS PROVIDED FOR ADAPTING IT TO YOUR HARDWARE.

1. CHANGING THE NUMBER OF AVAILABLE TRACKS--
   TO ALTER FOR 35 TRACKS ENTER THE FOLLOWING:
   ```
   POKE 17554,35 : CMD"M",0
   ```
   THIS WILL SET THE TRACK LIMIT TO 35 AND WRITE THE MODIFIED DOS TO DRIVE 0.

2. CHANGING THE STEPPING TIME--
   TO CHANGE THE STEP TIME TO 40 MILLISECONDS ENTER THE FOLLOWING:
   ```
   POKE 17556,3 : CMD"M",0
   ```
   THIS WILL CHANGE THE STEP TIME AND WRITE THE MODIFIED DOS TO DRIVE 0.

IMPORTANT NOTE

DISK DRIVES SOLD BY RADIO SHACK ARE CAPABLE OF ONLY 35 TRACKS OF STORAGE AND WILL NOT STEP FASTER THAN 40 MILLISECONDS. IF YOU ARE USING MICRODOS ON A SYSTEM WHICH CONTAINS ONE OR MORE RADIO SHACK DISK DRIVES YOU MUST APPLY THE ABOVE PATCHES FOR PROPER OPERATION.

* SIEMENS DISK DRIVES WERE FORMERLY SOLD AS WANGCO/PERKIN-ELMER

PERCOM DATA COMPANY JULY 11, 1979
COPYRIGHT (C) 1979

MICRODOS (TM) PATCH PROGRAM

THE FOLLOWING PROGRAM IS INTENDED TO BE USED IN APPLYING CORRECTIVE PATCHES TO THE PERCOM MICRODOS (TM) DISK OPERATING SYSTEM. ENTER IT CAREFULLY AS TYPOGRAPHICAL ERRORS COULD CAUSE VERY UNDESIRABLE RESULTS.

```
  100 CLEAR 1000
  110 CLS : PRINT "PERCOM PATCH UTILITY VERSION 1.00"
  120 PRINT "COPYRIGHT (c) 1979 PERCOM DATA COMPANY"
  130 PRINT
  200 PRINT "PHASE 1 - PATCH VALIDATION"
  210 LN=2000 : EC=0
  220 CK=0
  230 GOSUB 1000 : IF VL=0 THEN 300
  240 GOSUB 1000 : GOSUB 1000
  250 FOR I=1 TO VL : GOSUB 1000 : NEXT
  260 CK=-CK : GOSUB 1000
  270 IF CK=0 THEN 290
  280 PRINT "ERROR IN PATCH LINE";LN : EC=EC+1
  290 LN=LN+1 : GOTO 220
  300 IF EC>0 THEN PRINT "PATCHING ABORTED DUE TO ERRORS" : END
  310 RESTORE : PRINT "PHASE 2 - PATCH APPLICATION"
  320 READ VL : IF VL=0 THEN 500
  330 AD=VL*256 : READ VL : AD=AD+VL : IF AD>32768 THEN AD=AD-65536
  340 READ LN
  350 FOR I=1 TO LN
  360 READ VL
  370 POKE AD-1+I,VL
  380 NEXT I
  390 READ CK : GOTO 320
  500 PRINT "PHASE 3 - REWRITE SYSTEM"
  510 INPUT "PUSH 'ENTER' WHEN SYSTEM DISK ON DRIVE 0";ES
  520 CMD"M",0 : END
 1000 READ VL : CK=CK+VL : IF CK>255 THEN CK=CK-256
 1010 RETURN
```

PERCOM DATA CO.
COPYRIGHT (c) 1979
PATCH MEMO

PM-TFD-100-001
TFD-100 FLOPPY DISK
JULY 11, 1979

PRODUCT: PERCOM MICRODOS DISK OPERATING SYSTEM VERSION 1.10

PROBLEMS SOLVED:

1. IN 'LOAD'ING AND 'MERGE'ING PROGRAMS MICRODOS DOES NOT TAKE INTO CONSIDERATION THE FACT THAT THE PROGRAM MAY EXCEED THE MEMORY CAPACITY OF THE COMPUTER. IF SUCH IS THE CASE THE SYSTEM WILL MALFUNCTION UNPREDICTABLY.

PATCH APPLICATION

TO APPLY THE PATCHES ON THIS MEMO IT IS FIRST NECESSARY TO LOAD THE MICRODOS PATCH UTILITY PROGRAM INTO MEMORY. THIS PROGRAM IS PROVIDED ON SECTOR 40 OF THE MICRODOS SYSTEM DISKETTE FOR ALL COPIES OF MICRODOS VERSION 1.11 OR LATER. A COPY OF THE PROGRAM IS ALSO INCLUDED WITH THIS PATCH MEMO. IF ENTERING THE PATCH UTILITY PROGRAM BY HAND, BE ESPECIALLY CAREFUL, AS ERRORS COULD HAVE VERY UNDESIRABLE RESULTS.

ONCE THE PATCH UTILITY HAS BEEN LOADED THE PATCH LINES MUST BE ENTERED MANUALLY. WORK SLOWLY AND CAREFULLY TO AVOID MAKING MISTAKES. WHEN ALL PATCH LINES HAVE BEEN ENTERED TYPE 'RUN' TO APPLY THE PATCHES. THE PATCH UTILITY WILL DISPLAY A HEADING AND THE MESSAGE 'PHASE 1 - PATCH VALIDATION'. DURING THIS PHASE THE PATCH LINES ARE EXAMINED FOR POSSIBLE TYPING ERRORS. IF ANY ERRORS ARE FOUND THE OFFENDING LINE NUMBER IS DISPLAYED ALONG WITH THE MESSAGE 'PATCHING ABORTED DUE TO ERRORS'. THE LINE IN ERROR SHOULD BE CORRECTED AND THE PROGRAM 'RUN' AGAIN. ONCE ALL LINES HAVE BEEN VERIFIED CORRECT THE PATCH UTILITY WILL DISPLAY 'PHASE 2 - PATCH APPLICATION'. AT THIS TIME THE PATCHES ARE ACTUALLY APPLIED TO THE MICRODOS CURRENTLY RESIDING IN MEMORY. WHEN THIS PROCESS IS COMPLETE THE MESSAGE 'PHASE 3 - REWRITE SYSTEM' WILL BE DISPLAYED. THE UTILITY WILL THEN PROMPT 'PUSH ENTER WHEN SYSTEM DISK ON DRIVE 0'. AT THIS TIME LOAD THE DISK WHICH IS TO RECEIVE THE UPDATED COPY OF MICRODOS INTO DISK DRIVE 0 AND PUSH 'ENTER'. THE PATCHED MICRODOS WILL BE WRITTEN TO THAT DISK AND THE UTILITY WILL RETURN TO THE 'READY' PROMPT IN BASIC. IF OTHER DISKS ARE TO BE PATCHED THEY MAY BE UPDATED AT THIS USING THE 'CMD"M"' COMMAND TO WRITE THE UPDATE DOS ON THEM.

PATCH LINES - ENTER EXACTLY AS SHOWN

```
2000 DATA 73,205,3,205,140,85,199
2001 DATA 74,12,1,58,145
2002 DATA 74,66,5,205,93,27,24,193,175
2003 DATA 85,140,18,34,144,68,58,161,64,61,61,188,208,205,77,27,
          30,12,195,162,25,231
2004 DATA 69,67,1,49,186,Ø
```

DOCUMENTATION CHANGES

WITH THE APPLICATION OF THESE PATCHES YOUR COPY OF NICRODOS IS ELEVATED TO VERSION 1.11. NO OTHER DOCUMENTATION CHANGES ARE NECESSARY. PLEASE REFER TO THIS VERSION IN ANY CORRESPONDENCE RELATIVE TO MICRODOS.

PERCOM DATA CO.
COPYRIGHT (c) 1979
PATCH MEMO

PM-TFD-100-002
TFD-100 FLOPPY DISK
JULY 11, 1979

PRODUCT: PERCOM MICRODOS DISK OPERATING SYSTEM VERSION 1.11

PROBLEMS SOLVED:

1. IN 'SAVE'ING PROGRAMS MICRODOS WILL, IN RARE INSTANCES, REPORT THE LAST SECTOR USED AS ONE LESS THAN IT SHOULD. THIS COULD CAUSE UNDESIRED CONCATENATION OF PROGRAMS, OR THE APPENDING OF GARBAGE TO A PROGRAM WHEN IT IS 'LOAD'ED.

PATCH APPLICATION

TO APPLY THE PATCHES ON THIS MEMO IT IS FIRST NECESSARY TO LOAD THE MICRODOS PATCH UTILITY PROGRAM INTO MEMORY. THIS PROGRAM IS PROVIDED ON SECTOR 40 OF THE MICRODOS SYSTEM DISKETTE FOR ALL COPIES OF MICRODOS VERSION 1.11 OR LATER. A COPY OF THE PROGRAM IS ALSO INCLUDED WITH THIS PATCH MEMO. IF ENTERING THE PATCH UTILITY PROGRAM BY HAND, BE ESPECIALLY CAREFUL, AS ERRORS COULD HAVE VERY UNDESIRABLE RESULTS.

ONCE THE PATCH UTILITY HAS BEEN LOADED THE PATCH LINES MUST BE ENTERED MANUALLY. WORK SLOWLY AND CAREFULLY TO AVOID MAKING MISTAKES. WHEN ALL PATCH LINES HAVE BEEN ENTERED TYPE 'RUN' TO APPLY THE PATCHES. THE PATCH UTILITY WILL DISPLAY A HEADING AND THE MESSAGE 'PHASE 1 - PATCH VALIDATION'. DURING THIS PHASE THE PATCH LINES ARE EXAMINED FOR POSSIBLE TYPING ERRORS. IF ANY ERRORS ARE FOUND THE OFFENDING LINE NUMBER IS DISPLAYED ALONG WITH THE MESSAGE 'PATCHING ABORTED DUE TO ERRORS'. THE LINE IN ERROR SHOULD BE CORRECTED AND THE PROGRAM 'RUN' AGAIN. ONCE ALL LINES HAVE BEEN VERIFIED CORRECT THE PATCH UTILITY WILL DISPLAY 'PHASE 2 - PATCH APPLICATION'. AT THIS TIME THE PATCHES ARE ACTUALLY APPLIED TO THE MICRODOS CURRENTLY RESIDING IN MEMORY. WHEN THIS PROCESS IS COMPLETE THE MESSAGE 'PHASE 3 - REWRITE SYSTEM' WILL BE DISPLAYED. THE UTILITY WILL THEN PROMPT "PUSH ENTER WHEN SYSTEM DISK ON DRIVE 0'. AT THIS TIME LOAD THE DISK WHICH IS TO RECEIVE THE UPDATED COPY OF MICRODOS INTO DISK DRIVE 0 AND PUSH 'ENTER'. THE PATCHED MICRODOS WILL BE WRITTEN TO THAT DISK AND THE UTILITY WILL RETURN TO THE 'READY' PROMPT IN BASIC. IF OTHER DISKS ARE TO BE PATCHED THEY MAY BE UPDATED AT THIS USING THE 'CMD"M"' COMMAND TO WRITE THE UPDATE DOS ON THEM.

PATCH LINES - ENTER EXACTLY AS SHOWN

```
2000 DATA 73,105,3,205,158,85,117
2001 DATA 85,158,9,183,192,125,60,200,241,195,119,73,104
2002 DATA 69,67,1,50,187,Ø
```

DOCUMENTATION CHANGES

WITH THE APPLICATION OF THESE PATCHES YOUR COPY OF NICRODOS IS ELEVATED TO VERSION 1.12. NO OTHER DOCUMENTATION CHANGES ARE NECESSARY. PLEASE REFER TO THIS VERSION IN ANY CORRESPONDENCE RELATIVE TO MICRODOS.

PerCom Data Company, Inc.

11/26/79

A Procedure for Copying Programs from TRSDOS (tm) Diskettes to MICRODOS (tm) Diskettes

The following steps provide a simple means to copy any program in BASIC from a TRSDOS (tm) diskette to a MICRODOS (tm) diskette.

1. Load the MICRODOS (tm) operating system using either power-on or the RESET button. Insert a blank diskette in drive 0 and type the following commands:

```
CMD"K",""
CMD"I",0
```

   The diskette in drive 0 will be initialized to hold the program being copied.

2. Put the TRSDOS (tm) diskette in drive 0 and push the RESET button. Then load Disk BASIC in the normal fashion.

3. Take a piece of paper and write the letters A, B, C, and D in a column.

4. Load the BASIC program to be copied. Then type the following commands:

```
PRINT "A=";PEEK(16548)
PRINT "B=";PEEK(16549)
PRINT "C=";PEEK(16633)
PRINT "D=";PEEK(16634)
```

   Values will be printed on the screen for A, B, C, and D. Copy the values to your paper next to the letters already written there.

5. Load the previously initialized MICRODOS (tm) diskette in drive 0 and push the RESET button. When BASIC is 'READY', using your scratch paper, type in the following commands:

```
POKE 16548,(Value on paper for A)
POKE 16549,(Value on paper for B)
POKE 16633,(Value on paper for C)
POKE 16634,(Value on paper for D)
```

6. The program to be copied is now in memory. You may verify this by LISTing it. It may now be put on a MICRODOS (tm) diskette using the SAVE command.

   That's all there is to it! If you are going to continue to run in MICRODOS (tm), it is a good idea to push RESET after saving the program. This is necessary to restore the usable memory to the MICRODOS (tm) size, as it is actually "shrunk" for the copy procedure.