JDOS DISK BASIC REFERENCE MANUAL

J & M SYSTEMS, LTD.

September 1984

15100A Central Ave. SE • Albuquerque, New Mexico 87123 • (505)292-4182

JDOS EXTENDED DISK BASIC

for the

Radio Shack Color Computer

REFERENCE MANUAL

Revision C

September 1984

Copyright (c) 1983, 1984 by J & M Systems, Ltd.
15100 Central Ave S.E.
Albuquerque, NM 87108

DISCLAIMER

J & M Systems, Ltd. does not assume any responsibility or liability for loss or damages caused or alleged to be caused, directly or indirectly from the use or misuse of this manual or the associated software, including, but not limited to: any interruption of service, loss of business, anticipatory profits, or consequential damages. While every effort has been made to provide error-free software and documentation, J & M Systems assumes no responsibility for the consequences of the use or misuse of this or any other software and documentation. The user is the sole judge of his or her skill and ability to install and operate this software, and to make any modifications to any equipment or software.

Revision A B C

1st printing 1083
2nd printing 0984

JDOS Extended Disk BASIC V 1.11

Reference Manual

September 1984

Table of Contents

CHAPTER 1: Introduction to Extended Disk BASIC 1

CHAPTER 2: JDOS Extended Disk BASIC Instructions 7

Appendix A: JDOS Extended Disk BASIC Instruction Summary . . A-1

Appendix B: Interfacing to the DSKCON Subroutine B-1

Appendix C: Error Codes and Messages C-1

Appendix D: Installation of the J&M Systems Disk Controller. D-1

Appendix E: Disk Controller Pinouts. E-1

Appendix F: Interfacing Machine-Language Programs to JDOS. . F-1

This page intentionally blank

CHAPTER 1: INTRODUCTION TO JDOS EXTENDED DISK BASIC

SCOPE AND REFERENCES

This chapter presents some preliminary concepts concerning JDOS Extended Disk BASIC. It is assumed that the reader has a previous knowlege of the BASIC Language, as well as a basic understanding of disk systems. If this is not the case, you may wish to read the manuals listed below prior to using this manual. This manual is intended as a reference, and is concerned only with describing the additional instructions and capabilities provided by JDOS Extended Disk BASIC.

Radio Shack: "Getting Started with Color BASIC"
Radio Shack: "Going Ahead with Extended Color BASIC"
Radio Shack: "Color Computer Disk System:
Owner's Manual & Programming Guide"

Additional manuals are available from Radio Shack and other sources which describe the BASIC Programming Language in general.

JDOS EXTENDED DISK BASIC DISK CAPACITY

The data storage capacity of your JDOS Extended Disk BASIC Disks depends on what type of disk drive and what type of floppy disk controller you are using. The following table summarizes disk capacities for Radio Shack drives and controllers as well as drives and controllers available from J&M Systems, Ltd:

CONTROLLER & DRIVE	TRACKS	SECTORS PER TRACK	BYTES PER SECTOR	TOTAL CAPACITY (BYTES)
Radio Shack Controller:				
Color Disk #Ø (26-3Ø22)	35	18	256	161k
Color Disk #1,2,3 (26-3Ø23)	35	18	256	161k
J&M Systems Coco:				
4Ø-track, Single-sided	4Ø	18	256	184k
4Ø-track, Double-side	4Ø	36	256	368k

FILE NAMES

JDOS Extended Disk BASIC uses file names to keep track of files on disk. File names consist of two parts: the file name (fn) and the file name extension (/ext). A slash ("/") is used to separate the file name from the extension. So, the file name takes the form:

"fn/ext"

The file name must not be more than 8 characters long. Any characters may be used except "/". The extension is optional, but if you use an extension, it must be separated from the file name by a slash ("/").

The extension must not be more than 3 characters in length. The purpose of the extension is to indicate the file type. For example, a BASIC Program file usually has the extension "/BAS". A Data File usually has the extension "/DAT". A text file might have the extension "/TXT", and so on. So, when you list the directory of a disk, the file extensions give you a clue as to what types of files are on the disk.

Some file extensions are assigned by the computer. For example, if you save a BASIC Program to disk with the "SAVE" Command, and you do not specify a file extension on the command line, the computer will automatically assign the extension "/BAS" (refer to the description of the LOADM and OPEN Statements in Chapter 2 for other default assignments). Some of the more common file extensions used in Extended Disk BASIC are listed here:

EXTENSION	USED FOR
"/BAS"	BASIC Program files
"/DAT"	BASIC Data files
"/BIN"	Machine code program files
"/TXT"	Text files (as in word processing text)
"/FOR"	FORTRAN Source code files
"/REL"	Re-locatable object code files (assembler)

DISK FORMAT

The format of data stored on a JDOS Extended Disk BASIC disk is outlined in this section.

Track Format

A JDOS disk is divided into 35 or 40 tracks, numbered Ø-34 or Ø-39 (see the DSKINI Command, Chapter 2). Each track contains 18 sectors if the disk is single-sided, or 36 sectors if the disk is double-sided. For a detailed description of the track format, refer to the data sheet for the Western Digital FD-179X-Ø2 Floppy Disk Controller.

Sector Format

Each track is divided into 18 sectors (single-sided) or 36 sectors (double-sided), numbered 1-18 or 1-36. Each sector consists of a total of 338 bytes: 82 bytes of overhead and 256 bytes of data. The sector format is described in the FD-179X-Ø2 Data Sheet.

Directory Format

The disk directory is maintained on Track 17. The entire track is reserved for the directory, although only sectors 2 through 11 are actually used. Sector 2 contains the File Allocation Table, while Sectors 3-11 contain the actual directory entries. Up to 72 directory entries are allowed, and each entry is 32 bytes in length.

Each directory entry is organized as follows:

BYTE(s)	CONTENTS
Ø-7	File Name (fn)
8-1Ø	File Extension (/ext)
11	File Type
12	ASCII Flag
13	First granule number
14-15	Number of bytes used in last sector
16-31	Not used

File Name

The File Name is stored in ASCII in bytes Ø-7. It is left-justified in the field, and padded with ASCII spaces (2Ø Hex) if required. A delete mark of ØØ is used in the first byte of the field. That is, if the first byte of the file name is ØØ, the directory entry is marked for deletion, and may be used for new entries.

File Extension (/ext)

The File Extension (/ext) is stored in ASCII in bytes 8-1Ø. The "/" is not stored. The extension is left-justified, and padded with ASCII Spaces (2Ø Hex) if required.

File Type

The file type is a single-byte code in Byte 11 of the directory entry. The code is defined as follows:

CODE	MEANING
ØØ	BASIC Program File
Ø1	BASIC Data File
Ø2	Machine Code Program
Ø3	Text Editor Source File

ASCII Flag

The ASCII Flag is a code stored in Byte 12 of the directory entry. If the ASCII Flag is ØØ, the file is in "binary" (non-ASCII) format. If the ASCII Flag is FF Hex, the file is in ASCII format.

First Granule Number

A Granule is a basic file allocation block. All files are allocated an integral number of granules, so a file may occupy no less than one allocated granule.

A Granule consists of 9 sectors on a single-sided disk, or 18 sectors on a double-sided disk. Granule numbers are assigned sequentially beginning at Track Ø, Sector 1. On a 35-track single OR double sided disk there are 68 granules numbered Ø-67. On a 4Ø-track single OR double sided disk there are 78 granules numbered Ø-77.

Number of Bytes Used in Last Sector

This code, stored in Bytes 14 and 15 of the directory entry, represents the number of bytes used in the last sector in which the file is saved. The number is a 2-byte binary number in the range of ØØØØ - FFFF Hex.

THE FILE ALLOCATION TABLE

The File Allocation Table is located in Sector 2 of the directory track. This table is a map of allocated granules on the disk. The bytes in the sector correspond one-for-one with granule numbers. That is, Byte Ø corresponds to Granule Ø, Byte 1 corresponds to Granule 1, and so on. The last byte used corresponds to the highest granule number on the disk: 67 for single-sided 35-track disks, 77 for single-sided 4Ø-track disks, and 155 for double-sided 4Ø-track disks.

The code stored in each byte is defined as follows: If the code is FF Hex, the granule is unallocated. An unallocated granule is free for use, it does not belong to any file on the disk. If the code is in the range of ØØ - 9B Hex, the corresponding granule is allocated, and the code represents the number of the next granule allocated to the file. If the code is in the range of CØ - C9 Hex, the granule is allocated, and is the last granule in a file. Bits 6 and 7 represent the last granule flag, while bits Ø-5 represent the used sector count of the last granule in the file.

Sector Mapping

During formatting, a sector skew of 4 is applied to the disk. This is intended to minimize the time required to save or load a sequential file. The Logical-to-physical sector mapping is outlined in the table below:

PHYSICAL SECTOR	LOGICAL SECTOR	PHYSICAL SECTOR	LOGICAL SECTOR
1	1	1Ø	1Ø
2	12	11	3
3	5	12	14
4	16	13	7
5	9	14	18
6	2	15	11
7	13	16	4
8	6	17	15
9	17	18	8

This page intentionally blank

CHAPTER 2: JDOS EXTENDED DISK BASIC INSTRUCTIONS

JDOS Extended Disk BASIC provides a set of instructions which augment the existing instructional vocabulary of the Color Computer BASIC Interpreter. Most of the new instructions are used to control the disk system. All of the new instructions provided by JDOS are described in this chapter. Each instruction is described on a separate page for quick reference.

The descriptions of the JDOS Instructions are arranged in alphabetical order in this chapter. At the top of each page is the instruction name along with the word "Command", "Function", or "Statement". As used here, a "Command" is an instruction which may be used in direct mode (calculator mode). A "Statement" is normally used only in program mode. A "Function" is an instruction which returns a value, and may be used in either direct or program mode.

The description of each instruction includes the following three sections:

Syntax:

The Syntax section indicates the correct format for the instruction as well as all parameters, whether required or optional.

Purpose:

This paragraph explains what the instruction is used for, and provides some background to help you understand how and when to use the instruction.

Example:

This section provides one or more examples which illustrate typical applications of the instruction.

KEY WORDS AND SYMBOLS

Certain key words, abbreviations, and symbols are used consistently throughout the remainder of this manual to indicate instruction parameters. Some of the more commonly used parameters are symbolized as follows:

SYMBOL	MEANING	EXAMPLES
fn	File Name	MYPROG, FILE
/ext	File Name Extension	/BAS, /DAT
:d	Drive Number	:Ø, :1
#buffer	Buffer No. preceded by "#"	#1, #5
var	Variable	A, A1, B$
data	Constant or variable	A, 1, "STRING"

Most instructions include parameters which modify the operation of the instruction. The following notation conventions are used throughout this manual to describe parameters:

<parameter>

Parameters which must be included with an instruction are referred to in this manual as "required" parameters. A required parameter is denoted by the use of angle brackets ("<>") enclosing the parameter name. The angle brackets are not a part of the parameter, and they must not be included with the instruction. They are used only in the manual to indicate that a parameter is required.

[parameter]

Parameters which may be included with an instruction, but are not required, are referred to as "optional" parameters. An optional parameter is denoted by the use of brackets ("[]") enclosing the parameter name. The brackets are not a part of the parameter, and they must not be included with the instruction. They are used only in the manual to indicate that a parameter is optional.

",..."

A comma followed by three periods (",...") is used to indicate the optional continuation of a list. This symbology indicates that the immediately preceding parameter or parameter list may be continued indefinitely. Note, however, that all instruction lines are limited to a maximum of 255 characters.

Syntax:

AUTO [line][,increment]

Purpose:

The AUTO Command causes the computer to automatically generate BASIC program line numbers. When the AUTO Command is invoked, the computer will generate and display the first line number. Then, after each program line is entered, the computer will automatically generate and display the next line number.

Line numbers will be generated starting at line number [line], and the line numbers will increment by [increment]. If [line] is not specified, the first line will be numbered 10. If the [increment] is not specified, an increment of 10 will be used.

The AUTO Command is terminated by pressing the "ENTER" key immediately after a line number is displayed.

Example:

AUTO

AUTO 100,20

In the first example, the computer will generate line numbers beginning at 10 and incrementing by 10 (10, 20, 30, etc.). In the second example, the computer will generate line numbers beginning at 100 and incrementing by 20 (100, 120, 140, etc.).

Syntax:

BACKUP <source drive> TO <destination drive>

or

BACKUP <source drive>,<destination drive>

or

BACKUP Ø

Purpose:

The BACKUP Command is used to make a copy of a disk. The disk to be copied is referred to as the "source", since it is the source of the data to be copied. The disk which is to be the copy is referred to as the "destination", since it is the destination of the data to be copied. The BACKUP Command will copy the entire contents of the source disk to the destination disk.

The BACKUP Command will copy the entire contents of the source disk, sector-by-sector, to the destination disk. If there is any data on the destination disk, it will be written over. The destination disk must be formatted.

The action taken by the BACKUP Command will be different for multiple-drive systems than for single-drive systems:

USING THE BACKUP COMMAND ON MULTIPLE-DRIVE SYSTEMS:

Example:

BACKUP Ø TO 1

The computer will read data from the disk in Drive Ø and write it to the disk in Drive 1. This procedure will continue until the computer has read all of the sectors of the disk in Drive Ø, and written all of the data to the disk in Drive 1.

WARNING: The backup command will destroy data or programs in memory! DO NOT use this command if you have a program in memory that you want to keep! Use the SAVE Command to save the program prior to using the BACKUP Command.

USING THE BACKUP COMMAND ON SINGLE-DRIVE SYSTEMS:

Example:

BACKUP Ø

This is the only valid syntax for the BACKUP Command on systems with only one disk drive. Place the source disk in the disk drive, and enter the command "BACKUP Ø". Since the disk contains more data than the computer can hold in it's internal storage, you will have to help! First, the computer will read some data from the source disk into its memory. Then, the computer will instruct you to insert the destination disk in the disk drive:

INSERT DESTINATION DISKETTE AND PRESS ENTER

Insert the destination disk in the drive, close the drive door, and press the "ENTER" key. The computer will then write the data it has just read from the source disk to the destination disk. Then the computer will instruct you to place the source disk back in the drive:

INSERT SOURCE DISKETTE AND PRESS ENTER

Remove the destination disk from the disk drive, insert the source disk, close the drive door, and press the "ENTER" Key. The computer will then read some more data from the source disk.

This procedure will continue until all of the sectors on the source disk have been read into memory and written on to the destination disk. Be careful not to get the two disks mixed up during this procedure!

Syntax:

CLOSE [[#]buffer][,[#]buffer],...

Purpose:

The CLOSE Statement is used to terminate disk I/O operations. All disk I/O is handled through buffers (reserved areas in memory) assigned as needed with the OPEN Command.

When a disk read Statement (INPUT or GET) is executed, the required disk sector is read into the associated memory buffer. Subsequent INPUT statements will access data in the buffer until the requested data lies in another sector. Then a new sector may be read into the buffer. This technique increases the throughput of disk operations by minimizing disk accesses.

Disk write commands are handled in much the same way. Data to be written to disk are buffered in memory until the buffer is full, or until another sector is requested. At this point, the entire buffer will be written to the appropriate sector on the disk, thus minimizing disk I/O operations.

When an input file buffer is closed, the only action taken by the computer is to free the affected buffer space for subsequent use with another file. When an output file buffer is closed, the computer will write the buffer contents (if any) to the disk file, and update the directory entry to reflect the new file size. The output buffer space is then available for use with another file.

If a buffer number is specified with the CLOSE Command, only that buffer will be closed. If no buffer number is specified, all currently open buffers will be closed.

Example:

CLOSE 1, 3

CLOSE #1, #2

CLOSE

In the first example, only buffers 1 and 3 will be closed. In the second example, only buffers 1 and 2 will be closed. In the final example, all currently open buffers will be closed.

Syntax:

COPY <"fn1/ext[:d]"> TO <"fn2/ext[:d]">

or

COPY <"fn1/ext[:d]">,<"fn2/ext[:d]">

Purpose:

The COPY Command is used to make a duplicate copy of a file. The contents of the source file "fn1" are copied to the destination file "fn2". If the drive parameter [:d] is not specified, the current default drive will be used.

If the destination file already exists, the message:

FILE EXISTS. DELETE IT?

will be displayed. Respond by pressing the "Y" Key (for YES) if you want the existing file "fn2" to be deleted. Respond by pressing the "N" Key (for NO) if you do not want the existing file to be deleted. If you respond with an "N", the operation will be aborted, and the existing file will not be disturbed.

After the computer has completed execution of the COPY Command, the contents of the destination file will be identical to the contents of the source file, although the file names may be different.

Example:

COPY "MYPROG/BAS" TO "YOURPROG/BAS"

COPY "MYPROG/BAS:Ø" TO "MYPROG/BAS:1"

In the first example, the contents of the file "MYPROG/BAS" are copied to the file "YOURPROG/BAS". The source file must be on the default drive, and the destination file will be written on the default drive. In the second example, the file "MYPROG/BAS" on Drive Ø will be copied to the file "MYPROG/BAS" on Drive 1.

WARNING: In some cases, the COPY Command may destroy a program in memory. It is good practice to not use the COPY Command when there is a program in memory!

CAUTION: Be careful not to copy a file to a destination with the same name on the same disk. This will potentially result in the file being deleted!

USING THE COPY COMMAND ON SINGLE-DRIVE SYSTEMS

Example:

COPY "MYPROG/BAS" TO "MYPROG/BAS"

This is the only valid syntax for the COPY Command on systems with only one disk drive. When the command is entered, the computer will first prompt for the source disk:

INSERT SOURCE DISKETTE AND PRESS ENTER

Insert the disk with the source file on it into the drive, close the drive door, and press the "ENTER" Key. The computer will read some of the source file into memory and prompt for the destination disk:

INSERT DESTINATION DISKETTE AND PRESS ENTER

Insert the disk on which you want the copy written, close the drive door, and press the "ENTER" Key. The computer will write the file portion in memory to the destination disk. Depending on the length of the file, this sequence may repeat several times until all of the file has been copied.

Syntax:

CVN<(data)>

Purpose:

The CVN Function will decode a number which has been encoded into a string with the MKN$ (refer to the description of the MKN$ Function). The CVN Function will decode the 5-byte string code created by the MKN$ Function back into a number.

The CVN Function is normally used in conjunction with direct access file input since all numeric data stored in a direct access file must be converted into string data with the MKN$ Function. All numeric data read from a direct access file must be converted to numeric form with the CVN Function before it can be operated on as numeric data.

Example:

```
A = CVN (A$)
```

In this example, the variable "A" is assigned the numeric equivalent of the 5-byte encoded string A$. The encoded data in A$ was created by the MKN$ Function, or read from a direct-access file.

Syntax:

DIR[drive]

Purpose:

The DIR Command is used to display the directory of a disk. The directory of the disk in [drive] will be displayed. If the [drive] parameter is omitted, the directory of the disk in the current default drive will be displayed.

Example: DIRØ

```
MYPROG   BAS   Ø   B   3
FILE     DAT   Ø   A   4
TEST     BIN   1   B   1

SIDES = 1      TRACKS = 35R
```

The directory listing consists of a 5-column display. The information presented in each column is explained below:

COLUMN	MEANING	
1	File name	
2	File Extension	
3	File Type:	Ø => BASIC program file 1 => BASIC data file 2 => Machine language file 3 => Editor source file
4	Storage Format:	A => ASCII B => Binary
5	File Length (in sectors)	

At the bottom of the display, the number of formatted sides on the disk is indicated (SIDES = X, where X = 1 or 2). Also, the number of formatted tracks on the disk is indicated (TRACKS = XXY, where XX = 35 or 4Ø, and Y = "R" if the disk is formatted in Radio Shack-compatible format.

Syntax:

DOS

Purpose:

The DOS Command will load the "OS/9" Operating System from the disk in Drive Ø. The operating system is booted from Track 34 of the disk. The operating system will be loaded into memory, and control will be transferred to it. The disk in Drive Ø must be a standard Radio Shack version OS/9 boot disk.

Example:

DOS

Simply enter the command "DOS", and the OS/9 operating system will be booted from the disk in Drive Ø.

Syntax:

DRIVE <drive>

Purpose:

The DRIVE Command is used to change the default drive. When the computer is reset, the default drive will be Drive Ø. Drive Ø will remain the default drive until, and unless, you change the default drive with the DRIVE Command. The Drive number can only be changed to Ø, 1, or 2.

The default drive will be used for all commands where a drive number is required, except when a drive number is specified on the command line. Thus, the DRIVE Command is a convenience feature. If you are doing a lot of work on Drive 1, for example, you may use the command "DRIVE 1" to change the default drive to Drive 1. Then, you need not use the ":1" parameter on each command line.

The default drive will remain in effect until you change it, or until the computer is reset.

Example:

DRIVE 1

In this example, the default drive is changed to Drive 1. Drive 1 will remain the default drive until the DRIVE Command is used to change it, or until the computer is reset.

Syntax:

DSKINI<drive>

or

DSKINI<drive>,<sides>,<tracks>

Purpose:

The DSKINI Command is used to format a disk. A new disk must be formatted before it can be used by the computer. A used disk can be formatted to erase all of the data on the disk.

The first form of the DSKINI Command line is used to format a disk to be compatible with the Radio Shack disk format. This command will format the disk as 35 track, single-sided. Note, however, that the command "DSKINI0,1,35" does not provide compatibility with Radio Shack Operating System. The first form must be used for compatibility!

The second command format allows you to format a disk to match the characteristics of your disk drive: single- or double-sided, 35 or 40 tracks. The <drive> parameter specifies the number of sides (1 or 2) that are to be formatted, and the <tracks> parameter specifies the number of tracks (35 or 40) to be formatted.

Examples:

```
DSKINI1
DSKINI1,1,35
DSKINI0,2,40
```

In the first example, the disk in Drive 1 will be formatted to be compatible with the Radio Shack Operating System. In the second example, the disk in Drive 1 will be formatted as a single-sided, 35-track disk. In the final example, the disk in Drive 0 will be formatted as a double-sided, 40-track disk.

WARNING: The DSKINI Command will erase all of the data on the disk! The DSKINI Command also uses most of memory, so any program in memory will be destroyed! Do not use this command when you have a program in memory that you do not want to lose!

Syntax:

DSKI$ <drive>,<track>,<sector>,<var1$>,<var2$>

Purpose:

The DSKI$ Statement is a special disk I/O statement which allows data to be input directly from the disk. The DSKI$ Statement will input one sector (256 bytes) directly from the specified disk. The data will be input into two string variables, <var1$> and <var2$>. The string variable <var1$> will contain the first 128 bytes of the sector, while the string variable <var2$> will contain the second 128 bytes of the sector.

The DSKI$ Command requires the <drive>, <track>, and <sector> parameters. The <drive> parameter specifies the drive to be read (Ø-2). The <track> parameter specifies the track to read (Ø-39). The <sector> parameter specifies the sector to be read (1-18 for single-sided disks, or 1-36 for double-sided disks).

Example:

DSKI$ Ø,23,5,A1$,A2$

In this example, the data from Sector 5, Track 23 of the disk in Drive Ø will be read into variables A1$ and A2$. The first 128 bytes of the sector will be read into A1$, and the remaining 128 bytes will be read into A2$.

NOTE: Since JDOS is capable of handling several different disk formats, it is necessary that the DSKI$ routine "know" the format of the disk it is reading data from. This information is kept internally in JDOS and is updated during any directory access operation. Therefore, to ensure that this directory information is correct, it is recommended that some directory access statement be executed some time prior to execution of the first DSKI$ or DSKO$ Statement. For example, prior to executing the first DSKI$ Statement, execute a statement such as A=FREE(Ø).

Syntax:

DSKO$ <drive>,<track>,<sector>,<"data1">,<"data2">

Purpose:

The DSKO$ Statement is a special disk I/O statement which allows data to be written directly to the disk. The DSKO$ Statement will output up to 256 bytes directly to a specified sector. The first string <"data1"> will be written in the first half of the sector (bytes Ø-127), while the second string <"data2"> will be written on the second half of the sector (bytes 128-255). The data represented by <"data1"> and <"data2"> may be either string data or variables.

The DSKO$ Statement requires the parameters <drive>, <track>, and <sector>. The <drive> parameter specifies on which drive (Ø-2) the data is to be written. The <track> parameter specifies on which track (Ø-39) the data is to be written. The <sector> parameter specifies which sector is to be written (1-18 for single-sided disks, or 1-36 for double-sided disks).

CAUTION: This statement writes data directly to the disk, bypassing the file control system. Be careful not to destroy data on the disk!

Example 1:

```
DSKO$ Ø,15,1Ø,"STRING DATA 1","STRING DATA 2"
```

Example 2:

```
A$ = "STRING DATA 1"
B$ = "STRING DATA 2"
DSKO$ Ø,15,1Ø,A$,B$
```

In both examples, the string "STRING DATA 1" will be written on the first half of Sector 1Ø on Track 15 of the disk in Drive Ø. The string "STRING DATA 2" will be written on the second half of the sector.

NOTE: Refer to the NOTE near the bottom of the page describing the DSKI$ Statement for important additional information concerning the proper use of the DSKO$ Statement.

Syntax:

EOF <(buffer)>

Purpose:

The EOF Function returns the value -1 if there is no more data to be read from a file buffer. If there is more data, the value Ø is returned. So, the value returned by the EOF function can be tested to determine if all of a file has been read.

Example:

```
10 OPEN "I", #1, "TEST/DAT"

20 I = I + 1

30 IF EOF(1) = -1 THEN 80

40 GET #1, I

50 INPUT #1, A$

60 PRINT A$

70 GOTO 20

80 CLOSE #1
```

This program segment will read all of the data from the file "TEST/DAT". Each record read is printed until the end-of-file tag is encountered. At that point, the file will be closed.

Syntax:

ERL <(dummy argument)>

Purpose:

The Error Line Function (ERL) is used to determine the line number where an error was encountered during execution of a BASIC program. This function is most useful when used in conjunction with the ERROR Statement.

Note that although the ERL Function requires no argument, a dummy argument must be included on the Function Line to conform to Function syntax rules. Any argument may be used so long as it complies with syntax rules.

Example:

```
10     ERROR A=ERL(Ø):PRINT "AN ERROR WAS ENCOUNTERED ON
       LINE ",A," PLEASE CHECK YOUR PROGRAM FOR SYNTAX":GO
       TO 1ØØ
2Ø     HOURS=365*24
3Ø     PRINT HOURS
1ØØ    END
```

In this example, any error encountered during execution will cause execution to branch to Line 1Ø. At Line 1Ø, the line-number where the error occured will be displayed.

Syntax:

ERR <(dummy argument)>

Purpose:

The Error Number Function (ERR) returns an error code corresponding to the most recent error which occurred during execution of a BASIC Program. This function is most useful when used in conjunction with the error command.

Note that although the ERR Function requires no argument, a dummy argument must be included on the Function Line to conform to Function syntax rules. Any argument may be used so long as it complies with syntax rules.

The actual number returned by the ERR Function may be converted to the actual error code as follows:

Actual error code = (ERR/2)+1

where ERR is the number returned by the ERR Function.

Example:

```
10   N=N+1
20   ERROR IF ERL(0)=40 AND ERR(0)=52 THEN 60
30   FILE$="CHECK"+RIGHT$(STR$(N),1)
40   KILL FILE$
50   IF N<10 THEN 10 ELSE STOP
60   PRINT "THE FILE '",FILE$,"'  DOES NOT EXIST"
70   GO TO 50
```

In this example, any error occurring during execution of the program will cause execution to branch to Line 20. At Line 20, execution will branch to Line 60 if and only if the error occurred at Line 40 and the error was a "NE" Error (error code 27). Note that in this example, rather than convert the ERR Code to the actual error code, the value returned by ERR is simply compared to [(actual error code) - 1] * 2.

A list of all error codes and the corresponding error conditions is provided on the following page.

ERROR CODE	ERROR CONDITION
1	'NF' NEXT WITHOUT FOR
2	'SN' SYNTAX ERROR
3	'RG' RETURN WITHOUT GOSUB
4	'OD' OUT OF DATA
5	'FC' ILLEGAL FUNCTION CALL
6	'OV' OVERFLOW
7	'OM' OUT OF MEMORY
8	'UL' UNDEFINED LINE
9	'BS' BAD SUBSCRIPT
1Ø	'DD' REDIMENSIONED ARRAY
11	'/Ø' DIVISION BY ZERO
12	'ID' ILLEGAL DIRECT STATEMENT
13	'TM' TYPE MISMATCH
14	'OS' OUT OF STRING SPACE
15	'LS' STRING TOO LONG
16	'ST' STRING FORMULA TOO COMPLEX
17	'CN' CANNOT CONTINUE
18	'FD' BAD FILE DATA
19	'AO' ALREADY OPEN
2Ø	'DN' DEVICE NUMBER ERROR
21	'IO' INPUT OR OUTPUT ERROR
22	'FM' FILE MODE ERROR
23	'NO' FILE NOT OPEN
24	'IE' INPUT PAST END OF FILE
25	'DS' DIRECT STATEMENT IN FILE
26	'UF' UNDEFINED FUNCTION
27	'NE' FILE DOES NOT EXIST
28	'BR' BAD RECORD NUMBER
29	'DF' DISK FULL
3Ø	'OB' OUT OF BUFFER SPACE
31	'WP' DISK WRITE PROTECTED
32	'FN' BAD FILE NAME
33	'FS' BAD FILE STRUCTURE
34	'AE' FILE ALREADY EXISTS
35	'FO' FIELD OVERFLOW
36	'SE' SET TO NON-FIELDED STRING
37	'VF' VERIFICATION ERROR
38	'ER' WRITE OR INPUT PAST END OF FILE
39	'BI' BACKUP INCOMPATIBILITY
4Ø	'SR' STEP RATE ERROR
41	'BM' BAD MEMORY ERROR

Syntax:

ERROR < command line >

Purpose:

The ERROR Statement is used to trap errors encountered during execution of a BASIC program. The statement line consists of the word "ERROR" followed by any valid command line. The entire statement line must be less than 255 characters in length.

The ERROR Statement allows the programmer to issue an error message or take corrective action without suspending the program. If errors are to be trapped, the ERROR command must be executed before any errors are encountered in the program.

The error trap set by the ERROR Statement is reset each time an error is encountered. That is, after an error is encountered, the error trap is no longer in effect. If it is desired that the error trap remain in effect after each error is detected, then the ERROR Statement must be re-executed after each error occurs.

Example 1:

```
10   ERROR PRINT "THE FILE ENTERED DOES NOT EXIST":GOTO 1ØØ
2Ø   KILL "TEST/BAS:1"
3Ø   PRINT "THE FILE EXISTED AND WAS DELETED"
1ØØ  END
```

Example 2:

```
1Ø   N=N+1
2Ø   ERROR IF ERL(Ø)=4Ø AND ERR(Ø)=52 THEN 6Ø
3Ø   FILE$="CHECK"+RIGHT$(STR$(N),1)
4Ø   KILL FILE$
5Ø   IF N<1Ø THEN 1Ø ELSE STOP
6Ø   PRINT "THE FILE '",FILE$,"' DOES NOT EXIST"
7Ø   GO TO 5Ø
```

In Example 1, any error occuring during program execution will cause the message at line 1Ø to be displayed. In Example 2, an "NE" Error occuring after execution of Line 4Ø will cause line 6Ø to be executed. Note that the ERROR Statement is re-executed during each loop.

Syntax:

FREE<(drive)>

Purpose:

The FREE Function returns the amount of free (unused) disk space remaining on the disk in the specified drive. The <drive> parameter is required, and specifies which drive the computer is to check (Ø-2).

The FREE Function will return an integer value representing the number of unallocated sectors on the disk (each sector consists of 256 bytes of data). The following table indicates the total number of useable sectors available, not counting the directory, for the different disk formats:

TRACKS	SIDES	TOTAL SECTORS
35	1	612
35	2	1224
4Ø	1	7Ø2
4Ø	2	14Ø4

Example:

PRINT FREE(Ø)

In this example, the computer will print the number of free (unallocated) sectors remaining on the disk in Drive Ø.

Syntax:

```
GET <#buffer>[,record]
```

Purpose:

The GET Statement is used to read one record from a direct-access (random access) disk file to the associated input buffer. Note that the OPEN Statement must first be executed to associate an input buffer with a disk file.

The <#buffer> parameter is required, and specifies which buffer (1-15) is to receive the data. The <record> parameter is optional, and, if included, specifies which file record is to be read. If the <record> parameter is not specified, the next sequential record will be read. That is, the record number of the record currently in the buffer will be incremented by one, and the record corresponding to the new record number will be read into the buffer.

Example:

```
GET #1, 2

GET #3, N
```

In the first example, the computer will read Record Number 2 from the previously opened file into buffer #1. In the second example, Record Number "N" of the previously opened file is read into buffer #3. As shown in the second example, the record number may be specified through an integer variable. In this way, the record number to be read may be assigned under program control.

Syntax:

INPUT <#buffer>,<var1>[,var2][,var3],...

Purpose:

The INPUT Statement is used to move data from a disk input buffer to a program variable. Data items in the buffer are assigned to the variable names in the order that they are encountered on the INPUT Statement line. Variable <var1> will be assigned the first data item in the buffer, variable [var2] will be assigned the second data item, and so on.

The INPUT Statement requires a minimum of two parameters. The <#buffer> parameter specifies the buffer from which the data are to be obtained (Ø-15). The buffer number must be previously referenced in an OPEN Statement. The variable parameters (<var1>, [var2],...) specify the variable names into which the buffer data are to be moved. At least one variable name must be included on the INPUT Statement line.

The INPUT Statement requires a minimum of one variable name. As many variable names as required may be named on the command line, however the total line length may not exceed 255 characters total. The data from the buffer will be assigned to the variable names in the order that they appear on the command line.

Note that in the case of direct-access file input operations, the GET Statement must be executed prior to the INPUT Statement. Otherwise, there is not valid data in the direct-access record buffer, and the results of the INPUT Statement will be meaningless.

Example:

INPUT #1, A$, B$, C$

In this example, the data in Buffer #1, which has been previously opened, will be assigned to the variables A$, B$, and C$. The first data item in the buffer will be assigned to the variable "A$", the second data item will be assigned to the variable "B$", and the third datum will be assigned to the variable "C$".

Syntax:

KILL <"fn/ext[:d]">

Purpose:

The KILL Command is used to delete a file from a disk. The file name and extension must be specified. If the drive parameter [:d] is specified, the file named will be deleted from the disk in the specified drive. If the drive specification is omitted, the file named will be deleted from the disk in the current default drive.

Note that the file name, the extension, and the drive (if specified) must all be enclosed in quote marks. When the file is deleted, the disk space allocated to the file will be returned to the free-space pool providing more usable disk space.

Example:

KILL "MYPROG/BAS"

KILL "MYPROG/BAS:1"

In the first example, the file named "MYPROG/BAS" will be deleted from the disk in the current default drive. In the second example, the file named "MYPROG/BAS" will be deleted from the disk in Drive 1.

Syntax:

LINE INPUT <#buffer>,<var1$>

Purpose:

The LINE INPUT Statement is used to read a "line" of data from a disk I/O buffer. The buffer must be assigned to a file by a previously executed OPEN Statement, and there must be data in the buffer. A "line" of data is defined to be a group of data terminated by an "ENTER" character (ØD hex, 13 decimal). All data up to, but not including, the "ENTER" character will be transferred to the variable named on the statement line.

Two parameters are required in conjunction with the LINE INPUT Statement. The <#buffer> parameter specifies the input buffer (1-15) from which the data are to be taken. The <var1$> parameter specifies the string variable name into which the data are to be transferred.

Example:

LINE INPUT #1, B$

In this example, data from Input Buffer #1 will be transferred to the string variable "B$". All data up to the "ENTER" character will be transferred from the buffer to the string variable.

Syntax:

LIST [line number]

Purpose:

The LIST Command is used to list all or part of a BASIC Program in memory. If the [line number] parameter is not specified on the command line, the LIST Command will list the entire BASIC Program currently in memory, starting with the first line and stopping after the last line.

If the [line number] parameter is included, the LIST Command will list only the line number specified. The Up-arrow and Down-arrow Keys may then be used to list the line immediately before and after (respectively) the currently displayed command.

The Up-arrow and Down-arrow Keys may be used to scroll up and down (respectively) through the entire program. Each time the Up-arrow key is pressed, the next line up in the program (lower line number) will be displayed. Each time the Down-arrow key is pressed, the next line down in the program (higher line number) will be displayed.

Example:

```
LIST

LIST 5Ø
```

In the first example, the entire program currently in memory will be displayed on the screen. In the second example, Line 5Ø of the program currently in memory will be displayed on the screen.

Syntax:

LOAD <"fn[/ext][:d]">[,R]

Purpose:

The LOAD Command is used to transfer a BASIC program file from disk to main memory. When a program is loaded into memory, the program and data (if any) currently in main memory will be written over by the new program. Be careful not to load a file over a program that you want to keep!

The LOAD Command requires only one parameter: the file name. The file name extension is optional. If the file name extension is not included, the extension "/BAS" will be used.

The Drive parameter [:d] is also optional. If included, the file will be loaded from the disk in the drive specified. Otherwise, the current default drive will be used.

The "R" Parameter is an optional parameter. If included, the "R" parameter will instruct the computer to run the program immediately after it is loaded. This eliminates the need to enter the "RUN" Command after the file is loaded.

Example:

LOAD "MYPROG"

LOAD "MYPROG/BAS:1",R

In the first example, the file named "MYPROG/BAS" on the disk in the current default drive will be loaded into memory. In the second example, the file named "MYPROG/BAS" on the disk in Drive 1 will be loaded into main memory and will run immediately after it is loaded.

Syntax:

LOADM <"fn[/ext][:d]">[,offset]

Purpose:

The LOADM Command is used to transfer a machine code program from disk to main memory. When the LOADM Command is used, the program and/or data (if any) at the load address will be written over by the new program. Be careful not to destroy any program or data thet you wish to keep!

The LOADM Command requires only one patameter: the file name. The file name extension is optional. If the file name extension is not included, the computer will assume the extension of "/BIN".

The Drive parameter [:d] is also optional. If included, the file will be loaded from the disk in the drive specified. Otherwise, the current default drive will be used.

The <offset> parameter is an optional parameter. If included, the offset specified will be added to the program load address. The program will then be loaded at the address determined by the sum of the offset and the load address. Note that the offset parameter is assumed to be in decimal unless preceded by the characters "&H" to indicate that the number is hexadecimal.

Example:

LOADM "MYPROG"

LOADM "MYPROG/BIN:1",&H1000

In the first example, the file named "MYPROG/BIN" on the disk in the current default drive will be loaded into memory. The program will be loaded at the program's normal load address (as specified in the file). In the second example, the program named "MYPROG/BIN" will be loaded from the disk in Drive 1. The program will be loaded at the address determined by the sum of the program's normal load address and 1000 Hex.

Syntax:

LOC<(buffer)>

Purpose:

The LOC Function returns the record number of the file record currently in the specified buffer. It is assumed that a buffer has been previously opened (see OPEN Statement), and that a record has been placed in the buffer.

Only one parameter is required in conjunction with the LOC Function: the <buffer> parameter. This parameter specifies the buffer number (1-15) for which the record number is to be returned.

Example:

PRINT LOC(1)

A = LOC(3)

In the first example, the record number of the file record currently in Buffer #1 will be printed. In the second example, the record number of the file record currently in Buffer #3 will be assigned to the variable A.

Syntax:

LOF<(buffer)>

Purpose:

The LOF Function returns the last record number of a direct-access file. The specified buffer must be associated with a direct-access file name by a previously executed OPEN Statement. This function is especially useful when reading a file of unknown length.

The LOF Function requires only one parameter: the <buffer> parameter. This parameter specifies the buffer number (1-15) of the buffer associated with the file for which the last record number is to be returned. Note that the file name is associated with a buffer number only by the OPEN Statement, so the OPEN Statement must have been previously executed.

Example:

```
10 OPEN "D", #1, "EXAMPLE/TXT"

20 FOR R = 1 TO LOF(1)

30 GET #1, R

40 INPUT #1, E1$

50 PRINT E1$

60 NEXT R

70 CLOSE #1
```

In this example, the entire file will be read and printed. The FOR Loop in line 20 will increment R from 1 to the last record number of the file.

Syntax:

LSET <field> = <data>

Purpose:

The LSET Statement assigns data to the field name specified, and left-justifies the data. The field name must have been previously defined in a FIELD Statement. The data will be moved from the variable or direct assignment made in the LSET Statement <data> to the memory area reserved for the variable specified by the <field> parameter.

The LSET Statement will lift-justify the data as it is transferred to the field variable. If the data transferred is Xoo long for the field variable as defined in the FIELD Statement, the data will be truncated.

The LSET Statement may be used only with string data. If numeric data is to be LSET, it must be converted to string data first.

Example:

```
10 FIELD #1, 6 AS A1$, 10 AS A2$

20 LSET A1$ = "STRING 1"

30 LSET A2$ = B$
```

In this example, the field variable A1$ is assigned a length of 6 bytes, and A2$ is assigned a length of 10 bytes (both are defined by th FIELD Statement in line 10). In line 20, the data "STRING 1" is transferred and left-justified into the variable A1$. However, since the variable A1$ was assigned a length of 6 bytes, and the data "STRING 1" is 8 bytes in length, the data will be truncated. The result is that the variable A1$ will contain the data "STRING".

In line 30, the data contained in the varible "B$" will be transferred to, and left-justified in, the variable A2$. As in line 20, if the data contained in the variable "B$" is longer than 10 bytes (the assigned length of A2$), then A2$ will contain only the first 10 bytes of the data from B$.

Syntax:

MERGE <"fn/ext[:d]">[,R]

Purpose:

The MERGE Command is used to transfer a BASIC Program file from disk to memory, merging it with the program already in memory. Only a program file which was saved with the ASCII ("A") option may be merged (refer to the SAVE Command).

When two programs are merged, the result is that the program in memory will contain all of the lines from both programs. The only exception is that when there are line number conflicts (the same line number exists in both programs), the line from the disk file will replace the line already in memory.

The file name "fn" is required. The file name extension "/ext" is optional. If the file name extension is not specified, the extension "/BAS" will be used.

The drive parameter [:d] is also optional. If the drive parameter is not specified, the current default drive will be used.

The "R" parameter is optional. If included on the command line, the "R" Parameter will cause the program to be run immediately after the merge is complete.

Example:

MERGE "MYPROG"

MERGE "MYPROG/BAS:1",R

In the first example, the program file "MYPROG/BAS" will be merged with the program currently in memory. The program file must be on the disk in the current default drive. In the second example, the program file named "MYPROG/BAS" on Drive 1 will be merged with the program currently in memory. The resulting program will be run immediately after the merge is complete.

Syntax:

MKN$<(data)>

Purpose:

The MKN$ Function will convert a numeric variable, or a numeric constant, to a 5-byte coded string. This function is opposite to the CVN Function. The CVN Function may be used to decode the string back to a number.

The MKN$ Function is especially useful in direct access file operations, since all direct access file output must be in the form of string variables.

Example:

A$ = MKN$(123)

B$ = MKN$(C)

In the first example, the constant 123 will be converted to a 5-byte string code and stored in the variable "A$". In the second example, the number stored in the variable "C" will be converted to a 5-byte string code and stored in the variable "B$".

Syntax:

OPEN <"mode">,<#buffer>,<"fn[/ext][:d]">[,record length]

Purpose:

All disk file I/O is handled through buffers (reserved areas in memory). The OPEN Statement associates a specific disk file with a specific I/O buffer. The OPEN Statement must be executed prior to any file I/O (GET, PUT, INPUT#, PRINT#, etc.).

The "mode" parameter specifies the mode of operation for the file buffer:

"I" - Input from sequential access file
"O" - Output to sequential access file
"D" - Input and/or Output to/from direct access file

The <#buffer> parameter specifies the buffer number to be associated with the file named on the statement line. The <#buffer> parameter may be any number in the range of 1-15, but must not be the number of a buffer which is already open. Also, if buffer numbers other than #1 and #2 are to be used, the FILES Statement must be executed prior to the OPEN Statment.

The parameter "fn" specifies the file name to be associated with <#buffer> in all subsequent file I/O operations until the file is closed with the CLOSE Statement. If "/ext" is omitted, the extension "/DAT" will be used. If the drive parameter [:d] is omitted, the current default drive will be used.

The [record length] parameter is optional, and applies only to direct access files (mode "D"). If the [record length] is not specified, the record size will default to 256 bytes. This parameter may not be used with sequential access files.

Examples:

OPEN "I", #1, "TEST"
OPEN "D", #2, "DATA/TST:1", 40

In the first example, the file "TEST/DAT" on the disk in the current default drive is opened for input through Buffer #1. In the second example, the file "DATA/TST" on the disk in Drive 1 is opened for direct-access I/O through Buffer #2. The record length for this file is specified as 40 bytes.

Syntax:

PRINT <#buffer>,<data1>[,data2],...

Purpose:

The PRINT Statement moves the data named on the statement line to the disk output buffer named on the statement line. The data are formatted in the same way as when a normal PRINT Statement is executed. The only difference is that the data are sent to the disk file buffer rather than to the screen or the printer. Refer to the description of the PRINT Statement in the TRS-80 Disk System Owner's Manual for details.

Note that the data items named on the statement line may be separated by either commas (",") or semicolons (";"). As with the normal PRINT Statement, the comma will perform a tab function by inserting spaces between the data items. The semicolon will supress the spaces between data.

Example:

PRINT #1, "THIS IS A TEST", S

PRINT #5, A1$; A2$; N; L

In the first example, the string "THIS IS A TEST" and the contents of the variable "S" will be output to the disk file associated with Buffer #1. Because of the comma, a number of spaces will be output between the string and the contents of the variable "S". If a semicolon had been used instead of the comma, the spaces between the string and the data from the variable "S" would have been suppressed.

In the second example, the contents of the variables "A1$", "A2$", "N", and "L" will all be output to the disk file associated with Buffer #5. Since the variables are delimited by semicolons, no spaces will be output between the variables.

Syntax:

PRINT <#buffer>, USING <"format">;<data1>[,data2],...

Purpose:

The PRINT USING Statement moves the data named on the statement line to the disk output buffer named on the statement line. The data are formatted in the same way as when a normal PRINT USING Statement is executed. The only difference is that the data are sent to the disk file output buffer rather than to the screen or the printer. Refer to the description of the PRINT USING Statement in the TRS-80 Disk System Owner's Manual for more details.

The <format> parameter specifies the data format to be used when the data are output, in the same way as data are formatted when sent to the screen or the printer. The symbols used for formatting are summarized below.

#	Specifies a numeric field.
.	Places a decimal point position.
,	Places comma between each 3 digits in numeric field.
**	Fills leading spaces of a field with asterisks.
$	Places dollar sign at beginning of field.
$$	Places "floating" dollar sign adjacent to first digit in a numeric field.
+	If placed in first position of numeric field, indicates sign to be printed in front of number. If placed in last position of numeric field, indicates sign is to be printed after the number.
^	Indicates number to be printed in exponential format.
-	Places minus sign after negative numbers.
%	Delimits literal (string) fields.
!	Indicates use of only first character of a string.

Example:

PRINT #1, USING "###.##"; 197.664

This example would output the data "197.664" as "197.66". The third digit after the decimal point will be truncated since the specified format indicates only two decimal positions.

Example:

PRINT #3, USING "$$###.##-"; -3.78**

This example would result in the output of "**$3.78-" to the file associated with Buffer #3.

Example:

PRINT #1, USING "% %"; "STRING"

This example would output the data "STR" to the disk file associated with Buffer #1.

Syntax:

PUT <[#]buffer>[,record number]

Purpose:

The PUT Statement is used to write a direct access file buffer to the associated disk file. When performing direct access file output, the WRITE Statement transfers data to the disk buffer, and the PUT Statement transfers the buffer data to the disk file record.

The <buffer> parameter is required on the PUT Statement line, but the [record number] parameter is optional (note that the symbol "#" is optional in the <buffer> parameter). The [record number] parameter specifies the record number of the file to which the buffer is to be written. If this parameter is omitted, the record number which was last read into the buffer will be used.

Example:

```
10 OPEN "D", #3, "DATA/DAT", 10

20 FOR R = 1 TO 20

30 WRITE #1, "          "

40 PUT #3, R

50 NEXT R

60 CLOSE #3
```

This program segment will fill 10 records of the file "DATA/DAT" with ASCII spaces. Note that the file consists of 20 records of 10 bytes each.

Syntax:

RAM

Purpose:

The RAM Command will copy the contents of the Color Computer Read-Only Memory (ROM) to the read/write memory space (RAM). The contents of all of the ROM Memory from 8000 Hex to FFFF Hex will be copied into the RAM at the same addresses, and execution will continue in RAM. This allows patching and modification to the code, and provides additional expansion capabilities.

WARNING: The RAM Command will work only with a 64k system. On systems with less than 64k bytes of memory, the command will not work.

Example:

RAM

This command will move the contents of the ROM space (8000 Hex to FFFF Hex) to the same addresses in RAM memory. At the completion of the command, the computer will be executing the ROM code from RAM.

Syntax:

RATE <rate code>

Purpose:

The RATE Command is used to change the head step rate of the floppy disk controller. The following codes are used to select one of four step rates:

CODE	STEP RATE
Ø	6 ms.
1	12 ms.
2	2Ø ms.
3	3Ø ms.

When the computer is reset, the step rate will default to 6 ms. Do not select a step rate in excess of the maximum which your disk drives are capable.

Example:

RATE 3

In this example, the floppy disk step rate has been changed to 3Ø ms.

NOTE: Most of the old Model-I drives and the Radio Shack Color Computer drives cannot step at 6 ms. They must be set to the 3Ø ms rate. If your drives cannot step at the default 6 ms rate you must use this command to change the step rate before any disk access is attempted.

Syntax:

RENAME <"fn1/ext[:d]"> TO <"fn2/ext[:d]">

or

RENAME <"fn1/ext[:d]">,<"fn2/ext[:d]">

Purpose:

The RENAME Command is used to give an existing file a new name. The file contents will not be moved or changed in any way. Only the name of the file in the directory will be changed.

The "fn1" parameter specifies the current file name. The "fn2" parameter specifies the new file name. So, the RENAME Command will change the current file name "fn1" to the new file name "fn2"

The file name extension "/ext" is optional, but if the current file name "fn1" has an extension, it must be included on the command line.

The drive parameter [:d] is optional. If it is included, the file will be renamed on the disk in the drive specified. Otherwise, the file named is assumed to be on the disk in the current default drive.

Example:

RENAME "MYPROG/BAS" TO "YOURPROG/BAS"

RENAME "MYPROG:1", "YOURPROG/BAS:1"

In the first example, the file name "MYPROG/BAS" on the disk in the current default drive is renamed to "YOURPROG/BAS". In the second example, the file name "MYPROG" on the disk in Drive 1 is renamed to "YOURPROG/BAS".

Syntax:

RSET <field> = <data>

Purpose:

The RSET Statement assigns data to the field name <field>, and right-justifies the data. The field name <field> must have been previously defined in a FIELD Statement. The data will be moved from the variable or direct assignment <data> on the statement line to the memory area reserved for the variable <field>.

The RSET Statement will right-justify the data as it is transferred to the field variable. If the data transferred are too long for the field variable as defined in the FIELD Statement, the data will be truncated.

The RSET Statement may be used only with string data. If numeric data is to be RSET, it must first be converted to string data (refer to the MKN$ Function).

Example:

```
10 FIELD #1, 10 AS A1$, 5 AS A2$

20 RSET A1$ = "STRING 1"

30 B$ = "STRING 2"

40 RSET A2$ = B$
```

In this example, the variable "A1$" has been fielded as a 10-byte variable, and "A2$" has been fielded as a 5-byte variable (line 10). In line 20, variable "A1$" is RSET to "STRING 1". The result would be that A1$ = " STRING 1". Two spaces would be padded to the left of the data "STRING 1" so that the string is right-justified within the 10-byte field.

In line 40, A2$ is RSET to the variable B$ which contains the string "STRING 2". Since "STRING 2" consists of 8 bytes, and the variable A2$ has been fielded as a 5-byte variable (line 10), the contents of the field variable A2$ after execution of line 40 would be "STRIN".

Syntax:

RUN <"fn[/ext][:d]">

Purpose:

The RUN Command is used to load and run a BASIC program file from disk. The BASIC program file "fn" will be loaded from disk into memory and executed. If the extension parameter "/ext" is omitted, the extension "/BAS" is assumed. If the drive parameter [:d] is omitted, the program file is assumed to be on the disk in the current default drive.

Example:

RUN "MYPROG"

RUN "MYPROG/TST:1"

In the first example, the program saved in the file named "MYPROG/BAS" on the disk in the current default drive will be loaded and executed. In the second example, the program saved in the file named "MYPROG/TST" on the disk in Drive 1 will be loaded and executed.

Syntax:

RUNM <"fn[/ext][:d]">

or

..<fn1[/ext][:d]>

Purpose:

The RUNM Command is used to load and execute a machine-code program file. The program file "fn" will be loaded from disk into memory and executed. In the case of the first syntax, if the extension parameter "/ext" is omitted, the extension "/BIN" is assumed. If the drive parameter [:d] is omitted, the program file is assumed to be on the disk in the current default drive.

In the case of the second syntax, if the file name extension is omitted, the extension "/COM" is assumed. As in the first syntax, if the drive parameter [:d] is omitted, the current default drive is assumed.

Examples:

RUNM "MYPROG"

RUNM "MYPROG/TST:1"

..TSTPROG

In the first example, the machine-code program in the file named "MYPROG/BIN" will be loaded and executed. Since no drive parameter was specified, the program file must be on the disk in the current default drive. In the second example, the machine-code program in the file named "MYPROG/TST" on the disk in Drive 1 will be loaded and executed. In the final example, the machine-code program in the file "TSTPROG/COM" on the disk in the current default drive will be loaded and executed.

Syntax:

SAVE <"fn[/ext][:d]">[,A]

Purpose:

The SAVE Command is used to save a BASIC program to disk. The BASIC program currently in memory will be saved in a disk file with the name "fn" as specified on the command line. If the extension "/ext" is not included on the command line, the extension "/BAS" will be used. If the drive parameter [:d] is omitted from the command line, the program will be saved on the disk in the current default drive.

If the "A" parameter is included on the command line, the program will be saved in ASCII form on the disk. If the "A" parameter is omitted from the command line, the program will be saved in a coded binary form. Although a program saved in ASCII form will require more disk space, it may be edited like any other ASCII text file. Also, a program which is to be merged with another file must be saved in ASCII form (refer to the description of the MERGE Command).

If the file name specified on the command line already exists in the disk directory, the message:

FILE EXISTS. REPLACE IT?

will be displayed. Respond by pressing the "Y" Key (for Yes) if you wish to replace the existing file. If you press the "N" Key (for No), the existing file will not be disturbed, and the program will not be saved.

Example:

SAVE "MYPROG"

SAVE "MYPROG/TST:1",A

In the first example, the program currently in memory will be saved in a file named "MYPROG/BAS" on the disk in the current default drive. In the second example, the program currently in memory will be saved in ASCII Format in a file named "MYPROG/TST" on the disk in Drive 1.

Syntax:

SAVEM <"fn[/ext][:d]">,<first>,<last>,<execution>

Purpose:

The SAVEM Command is used to save a machine-code program or data to disk. The program or data currently in memory will be saved in a file named "fn". If the file name extension "/ext" is not specified, the extension "/BIN" will be used. If the drive parameter [:d] is not specified, the program will be saved on the disk in the current default drive.

The <first> parameter specifies the first (lowest) address of the program to be saved. The <last> parameter specifies the last (highest) address of the program. The <execution> parameter specifies the entry point (execution address) of the program. Note that the <first>, <last>, and <execution> address parameters are assumed to be in decimal unless they are preceded with "&H" which indicates that the numbers are hexadecimal.

If the file name "fn" specified on the command line already exists in the disk directory, the message:

FILE EXISTS. REPLACE IT?

will be displayed. Respond by pressing the "Y" Key if you wish to replace the existing file. If you press the "N" Key, the existing file will not be disturbed, and the program currently in memory will not be saved.

Example:

SAVEM "MYPROG",1024,2048,1024

SAVEM "MYPROG/TST:1", &H1000, &H1400, &H1020

In the first example, the program located in memory between adresses 1024 decimal and 2048 decimal will be saved in a file named "MYPROG/BIN" on the disk in the current default drive. The execution address (entry point) of this program is defined to be 1024 decimal. In the second example, the program in memory between 1000 Hex and 1400 Hex will be saved in a file named "MYPROG/TST" on the disk in Drive 1. The execution address of this program is specified to be 1020 Hex.

Syntax:

UNLOAD [drive]

Purpose:

The UNLOAD Command is used to close all files which are currently open on a specified drive. If you change disks while one or more files are open, it is very likely that you will destroy the directory on the disk. For this reason, it is good practice to use the UNLOAD Command before changing disks. Of course, if you are not doing any file I/O, you don't need to use the UNLOAD Command.

The [drive] parameter on the command line is optional. If the drive parameter is included, the files which are open on the drive specified will be closed. If the drive parameter is omitted from the command line, the files which are open on the current default drive will be closed.

Example:

UNLOAD

UNLOAD 1

In the first example, all of the files open on the current default drive will be closed. In the second example, all of the files open on Drive 1 will be closed.

Syntax:

VERIFY ON

or

VERIFY OFF

Purpose:

The VERIFY Command is used to control the disk I/O read-after-write verify function. Normally, the computer does not verify disk write operations. When the command "VERIFY ON" is invoked, the computer will perform read-after-write verification of all data written to disk. This verification function will remain in effect until either the command "VERIFY OFF" is executed, or the computer is reset.

When the computer performs a read-after-write verification, it will read a sector back just after it is written to disk. It will then compare what it read from the sector with what it wrote to the sector. If the two are not the same, the computer will attempt to write the sector again, up to a total of five times. If it still cannot record the data properly it will issue an I/O Error message. In this way, you may be sure that the data was written to disk properly, before it is too late.

Example:

VERIFY ON

In this example, the read-after-write verification function is enabled. It is good practice to use this feature at all times.

Syntax:

WRITE <#buffer>,<data1>[,data2],...

Purpose:

The WRITE Statement is used to transfer data from data constants or variables to a specified disk output buffer. The data items will be transferred to the buffer in the order that they appear on the statement line. If the specified buffer is associated with a direct-access file, the data items will be placed in the buffer in fields immediately adjacent to one another. Data items in a sequential-access file will be separated by commas.

The <#buffer> parameter specifies the buffer (1-15) to which the data is to be transferred. At least one data item must be present on the statement line. The data item may be either a constant or a variable. If more than one data item is specified, the items must be separated by commas (","). Unlike the PRINT Statement, the commas on the WRITE Statement line perform no function other than delimiting the items in the list.

Example:

WRITE #1, A, B, B$, "STRING"

In this example, the contents of the variables "A", "B", and "B$", as well as the string constant "STRING", will be transferred to disk output buffer #1. The contents of the variable "A" will be placed in the buffer first, followed by the contents of "B", the contents of "B$", and finally by the string constant "STRING".

APPENDIX A

JDOS Extended Disk Basic Instruction Summary

AUTO [line][,increment]
Example: AUTO 1ØØ,5Ø
Automatically generates BASIC Program line numbers starting at line number [line] and incrementing by [increment]. Terminate by pressing "ENTER" key immediately following any line number.

BACKUP <source drive> TO <destination drive>
BACKUP <source drive>,<destination drive>
BACKUP Ø
Example: BACKUP Ø TO 1
Copies entire contents of disk in <source drive> to disk in <destination drive>. First two forms for multiple-drive systems, third form for single-drive systems.
WARNING: This command will destroy programs or data in memory!

CLOSE [[#]buffer][,[#]buffer],...
Example: CLOSE 3, 2
Close disk files associated with buffer(s) specified. If no buffer numbers are specified, all open files will be closed.

COPY <"fn1/ext[:d]"> TO <"fn2/ext[:d]">
COPY <"fn1/ext[:d]">,<"fn2/ext[:d]">
Example: COPY "MYPROG/BAS" TO "MYPROG/BAS:1"
Copy contents of source file "fn1" to destination file "fn2".
WARNING: This command may destroy programs or data in memory!

CVN<(data)>
Example: A = CVN(A$)
Decodes a 5-byte, string-encoded variable to numeric form (see also MKN$).

DIR[drive]
Example: DIRØ
Displays directory of disk in specified drive. If [drive] is not specified, directory of disk in default drive is displayed.

DOS
Example: DOS
Loads and executes ("boots") the OS/9 Operating System from the disk in Drive Ø.

DRIVE <drive>
Example: DRIVE 1
Changes default drive number to <drive>. Default drive number will be used in any instruction where the drive number is not specified.

DSKINI<drive>,<R>
DSKINI<drive>,<sides>,<tracks>
Example: DSKINI1,1,40
Formats the disk in the drive specified. Specify <drive> (0-3), <sides> (1 or 2), <tracks> (35 or 40). First form for Radio Shack-compatible disk format only (single-sided, 35-track).
WARNING: This command will destroy programs or data in memory!

DSKI$ <drive>,<track>,<sector>,<var1$>,<var2$>
Example: DSKI$ 0,23,5,A1$,A2$
Inputs <sector> of <track> on disk in <drive> directly to <var1$> and <var2$>. String variable <var1$> will receive first 128 bytes of sector, <var2$> will receive second 128 bytes.

DSKO$ <drive>,<track>,<sector>,<"data1">,<"data2">
Example: DSKO$ 0,15,10,"STRING DATA 1","STRING DATA 2"
Writes <"data1"> and <"data2"> directly to <sector> of <track> on disk in <drive>. String constant or variable <"data1"> is written to first half of sector, <"data2"> is written to second half of sector.

EOF<(buffer)>
Example: IF EOF(1) = -1 THEN 60
Returns the value -1 if there is no more data to be read from a disk file. Returns value 0 if there is more data.

ERL <(dummy argument)>
Example: A = ERL(0)
Returns the line number where the last error occured during execution of a BASIC Program.

ERR <(dummy argument)>
Example: A = ERR(0)
Returns the error code associated with the last error which occurred during execution of a BASIC Program.

ERROR <command line>
Example: ERROR IF ERL(Ø)=4Ø and ERR(Ø)=52 THEN 6Ø
Traps errors during execution of a BASIC Program, allowing the program to take corrective action and preventing the interpreter from suspending execution.

FIELD <#buffer>,<field size> AS <field name>,...
Example: FIELD #1, 5 AS A1$, 1Ø AS A2$, 7 AS B$
Associates variable names with, and defines fields in disk file buffer. Parameter <#buffer> specifies buffer (1-15), <field size> specifies number of bytes to be reserved for variable <field name>. Fields are reserved in the order they are specified on the statement line.

FILES <number>[,size]
Example: FILES 5,1ØØØ
Reserves memory for <number> file buffers, and [size] bytes for direct-access record buffer. If [size] is not specified, 256 bytes total will be reserved. System defaults to FILES 2,256.

FLEX
Example: FLEX
Loads and executes ("boots") the FLEX Operating System from the disk in Drive Ø.

FREE<(drive)>
Example: PRINT FREE(Ø)
Returns number of unallocated granules on disk in drive <drive>. 1 granule = 9 sectors (23Ø4 bytes) on single-sided disks, 18 sectors (46Ø8 bytes) on double-sided disks.

GET <#buffer>,[record]
Example: GET #1, 2
Reads [record] from direct access file to <#buffer>. File must be open. If [record] not specified, next sequential numbered record will be read.

INPUT <#buffer>,<var1>[,var2],...
Example: INPUT #2, A$, A1$, C$
Transfers data from <#buffer> to variables <var1>, [var2],... in the order that the variables appear on the statment line.

KILL <"fn/ext[:d]">
Example: KILL "MYPROG/BAS"
Deletes file "fn/ext" from disk. If drive [:d] is not specified, current default drive will be used.

LINE INPUT <#buffer>,<var1$>
Example: LINE INPUT #1, B$
Transfers line of string data from <#buffer> to <var1$>. Data are transferred until "ENTER" character (ØD Hex) is encountered.

LIST [line number]
Example: LIST 5Ø
Lists the BASIC Program currently in memory. If [line number] is omitted, entire program will be listed. Otherwise, only the line specified will be listed, and up-arrow will list previous line, down-arrow will list next line.

LOAD <"fn[/ext][:d]">[,R]
Example: LOAD "MYPROG:1",R
Loads BASIC program from disk file "fn" to memory. If file name extension "/ext" is not specified, "/BAS" is used. If drive [:d] is not specified, current default drive is used. If "R" is specified, program will be run immediately after it is loaded.

LOADM <"fn[/ext][:d]">[,offset]
Example: LOADM "MYPROG:1",&H1ØØØ
Loads machine-code program "fn" from disk to memory. If file name extension "/ext" is not specified, "/BIN" is used. If drive [:d] is not specified, current default drive is used. Program is loaded into memory at (load address + [offset]). Load address is specified when program is saved. If [offset] is not specified, ØØØØ is used.

LOC<(buffer)>
Example: A = LOC(1)
Returns the record number of the record currently in <(buffer)>.

LOF<(buffer)>
Example: FOR R = 1 TO LOF(2)
Returns the highest (last) record number of the file associated with <buffer> (direct-access files only).

LSET <field> = <data>
Example: LSET A2$ = B$
Transfers data from variable or constant <data> to <field> variable. Data will be left-justified in <field> after transfer.

MERGE <"fn/ext[d:]">[,R]
Example: MERGE "MYPROG/BAS:1",R
Merges BASIC Program file "fn" with BASIC Program in memory. Program file to be merged must have been saved in ASCII form. If drive [:d] is not specified, current default drive will be used. If "R" is specified, program will be run immediately after merge.

MKN$<(data)>
Example: A$ = MKN$(C)
Converts numeric constant or variable <(data)> to 5-byte string data (see also CVN).

OPEN <"mode">,<#buffer>,<"fn/ext[d:]">[,record length]
Example: OPEN "I", #1, "TEST"
Opens disk file "fn" for I/O through <#buffer>. Parameter <"mode"> is "I" for sequential-access input, "O" for sequential-access output, "D" for direct-access input or output. Parameter <#buffer> must be 1-15, and must not be already open. If"/ext" is not specified, "/DAT" will be used. If [:d] is not specified, current default drive will be used. If [record length] is not specified, 256-byte records will be assumed (direct-access only).

PRINT <#buffer),<data1>[,data2],...
Example: PRINT #2, A1$, A2$; B; L
Transfers data from constants or variables <data> to <#buffer>. Data are transferred in the order that the <data> are named on the statement line.

PRINT <#buffer>, USING <"format">;<data1>[,data2],...
Example: PRINT #1, USING "###.##"; A
Transfers data from constants or variables <data> to <#buffer>. Data are transferred in the order that the <data> are named on the statement line.

PUT <[#]buffer>[,record number]
Example: PUT #3, R
Writes data in <[#]buffer> to [record number] of disk file associated with <[#]buffer>. If [record number] is not specified, record number of record currently in <[#]buffer> will be used.

RAM
Example: RAM
Copies contents of ROM memory to RAM Memory, continues execution from RAM Memory.
WARNING: This command will work only on 64k systems!

RATE <rate code>
Example: RATE Ø
Changes the floppy disk controller head step rate. Use rate code Ø for 6 ms step rate, 1 for 12 ms, 2 for 2Ø ms, and 3 for 3Ø ms. Default after reset is 6 ms.

RENAME <"fn1/ext[d:]"> TO <"fn2/ext[d:]">
RENAME <"fn1/ext[d:]">,<"fn2/ext[d:]">
Example: RENAME "MYPROG/BAS:1" TO "YOURPROG/BAS:1"
Renames file named "fn1" to new name "fn2". If [:d] is not specified, current default drive will be used.

RSET <field> = <data>
Example: RSET A2$ = B$
Transfers data from constant or variable <data> to field variable <field>. Data are right-justified in <field> after transfer.

RUN <"fn/ext[d:]">
Example: RUN "MYPROG"
Loads and runs BASIC program file "fn". If "/ext" is not specified, "/BAS" will be used. If [:d] is not specified, current default drive will be used.

RUNM <"fn/ext[d:]">
or ..fn/ext[d:]
Example: ..MYPROG
Loads and executes machine-code file from file "fn". If "/ext" is not specified, "/BIN" will be used in first form, "/COM" will be used in second form. If [:d] is not specified, current default drive will be used.

SAVE <"fn/ext[d:]">[,A]
Example: SAVE "MYPROG:1",A
Saves the BASIC program in memory to disk file "fn". If "/ext" is not specified, "/BAS" will be used. If [d:] is not specified, the current defauly drive will be used. If "A" is specified, the program will be saved in ASCII format.

SAVEM <"fn[/ext][:d]">,<first>,<last>,<execution>
Example: SAVEM "MYPROG", &H1000, &H1400, &H1200
Saves machine-code program in memory at address <first> through <last> to disk file "fn". Entry point address <execution> will be saved with file. If "/ext" is not specified, "/BIN" will be used. If [:d] is not specified, the current default drive will be used. Parameters <first>, <last>, and <execution> must be decimal unless preceded with "&H" to indicate hexadecimal.

UNLOAD [drive]
Example: UNLOAD 0
Closes all files currently open on [drive]. If [drive] is not specified, current default drive is assumed.

VERIFY ON
VERIFY OFF
Example: VERIFY ON
Turns ON or OFF the read-after-write verify option. Option remains in effect until changed or computer is reset. After reset, verify option is OFF.

WRITE <#buffer>,<data1>[,data2],...
Example: WRITE #1, A, B, "STRING"
Transfers data from constants or variables <data> to <#buffer>. Data items are transferred in the order that they are named on the statement line.

This page intentionally blank

APPENDIX B

INTERFACING TO THE DSKCON SUBROUTINE

Introduction

As a convenience to machine-code programmers, the basic disk I/O driver of the JDOS Extended Disk BASIC ROM is documented here. The disk I/O driver is called "DSKCON", and may be interfaced only on a machine-code level.

The DSKCON Subroutine is a table-driven disk I/O driver which provides the following primitive disk I/O functions:

- Select drive
- Restore (home) head
- Seek track
- Read sector
- Write sector
- Write track

Interface Addresses

The address of the Disk Parameter Table (DPB) is stored in memory location CØØ6 Hex. The entry point address of the DSKCON Subroutine is stored at memory address CØØ4 Hex.

Description

Prior to calling the DSKCON Subroutine, the Disk Parameter Table (DPB) must be set up. The DPB is defined as follows:

LABEL	#BYTES	OFFSET	DESCRIPTION	RANGE
FDCODE	1	Ø	Driver operation code	(ØØ-Ø7)
DRIVE	1	1	Requested disk drive	(ØØ-Ø2)
TRACK	1	2	Requested track	(ØØ-39)
SECTOR	1	3	Requested sector	(Ø1-36)
BUFAD	2	4	Buffer address	(ØØØØ-FFFF)
FDCSTA	1	6	Completion code	(ØØ-FF)
DVSTEP	1	1Ø	Step-rate code	(ØØ-Ø3)

J & M SYSTEMS, LTD.

The Driver Operation Code (FDCODE)

The Driver Operation Code is the first parameter in the DPB. This code defines the operation that the driver is to perform. The code must be in the range of 00-07 as defined below:

FDCODE	Operation performed
00	Home (restore) head to track 0
01	No operation
02	Read sector
03	Write sector
04	Write track
05	Select drive
06	Home head (same as 00)
07	Home head (same as 00)

The Drive Code (DRIVE)

The Drive Code (DRIVE) determines which disk drive DSKCON is to use for the requested operation. Use code 00 for Drive 0, 01 for Drive 1, and 02 for Drive 2.

The Track Code (TRACK)

The Track Code (TRACK) specifies which track DSKCON is to use for the requested operation. The track number must be in the range of 00-34 for 35-track formatted disks, or 00-39 for 40-track formatted disks.

The Sector Parameter (SECTOR)

The Sector Parameter (SECTOR) specifies which sector DSKCON is to use for the requested operation. The sector number must be in the range of 01-18 for single-sided disks, or 01-36 for double-sided disks.

The Buffer Address Parameter (BUFAD)

The Buffer Address Parameter (BUFAD) specifies the address of the buffer that DSKCON is to use for disk read/write operations. The buffer must be 256 bytes in length, from BUFAD to BUFAD+255.

The Completion Code Parameter (FDCSTA)

The Completion Code Parameter (FDCSTA) is an error code parameter returned by DSKCON at the completion of the requested operation. An error code of Ø ((FDCSTA)=ØØ) means that the operation was completed with no error. In the event of an error, DSKCON will return a bit-mapped code defined as follows:

FDCSTA BIT	MEANING	
Ø	Not used:	Always Ø
1	Not used:	Always Ø
2	Lost Data:	When set (1), the computer did not respond to a DRQ.
3	CRC Error:	When set (1), the FDC has detected a CRC error.
4	Seek Error:	When set (1), a track seek did not verify properly.
or	RNF:	When set (1), the requested record was not found.
5	Write Fault:	When set (1), the FDC has detected a write fault.
6	Write Prot:	When set (1), the disk is write-protected.
7	Not Ready:	When set (1), the requested drive is not ready.

The Drive Step Rate Parameter (DVSTEP)

The Drive Step Rate Parameter (DVSTEP) specifies the head step rate. Set DVSTEP to ØØ-Ø3 as shown below:

DVSTEP	STEP RATE
ØØ	6 ms.
Ø1	12 ms.
Ø2	2Ø ms.
Ø3	3Ø ms.

DISK I/O OPERATIONS

Home Head (Opcode ØØ)

This operation will restore the head of the requested drive to track Ø. This operation should be performed before the first seek to be sure that the FDC Track register contains the correct track number. Note that the driver maintains a track table for each drive, so it is not necessary to home the head each time a new drive is selected.

Example:

This example will restore the head of Drive Ø to track Ø:

```
HOME    LDX    #$CØØ6      POINT TO DPB
        CLR    ,X          OP CODE FOR HOME
        CLR    1,X         SELECT DRIVE Ø
        JSR    [$CØØ4]     CALL DSKCON
```

No Operation (Opcode Ø1)

This operation does not perform a read or write operation. However, it does perform all other operations that would be performed prior to a read or write operation. The requested drive is selected and the requested track is seeked. These operations are normally performed by all other Op codes, but this "No-op" operation is useful in cases where there is a need to keep drives spinnig, etc.

Read Sector (Opcode Ø2)

This operation will read one sector from the disk to memory. All parameters in the parameter table are required. If the head is not already over the requested track, the track will be seeked.

Example:

The following example will read Sector 5 of Track 1Ø into a buffer at 1ØØØ Hex:

```
READ    LDX     #$CØØ6      POINT TO DPB

        LDA     #Ø2         SET READ SECTOR OP CODE
        STA     ,X

        CLR     1,X         SELECT DRIVE Ø

        LDA     #1Ø         SELECT TRACK 1Ø
        STA     2,X

        LDA     #Ø5         SELECT SECTOR 5
        STA     3,X

        LDU     #$1ØØØ      SET BUFFER ADDRESS
        STU     4,X

        LDA     #Ø3         SET STEP RATE
        STA     1Ø,X

        JSR     [$CØØ4]     CALL DSKCON

        LDA     6,X         GET ERROR CODE
        BNE     ERROR       BRANCH ON ERROR
```

Write Sector (Opcode Ø3)

This operation will write one sector from memory to disk. All parameters in the parameter table are required. If the head is not already over the requested track, the track will be seeked.

Example:

The following example will write the contents of the buffer at 1ØØØ Hex to Track 2, Sector 19 of Drive 1:

```
WRITE   LDX     #$CØØ6      POINT TO DPB

        LDA     #Ø3         SET WRITE SECTOR OP CODE
        STA     ,X

        LDA     #Ø1         SELECT DRIVE 1
        STA     1,X

        LDA     #Ø2         SELECT TRACK 2
        STA     2,X

        LDA     #Ø5         SELECT SECTOR 5
        STA     3,X

        LDU     #$1ØØØ      SET BUFFER ADDRESS
        STU     4,X

        LDA     #Ø3         SET STEP RATE
        STA     1Ø,X

        JSR     [$CØØ4]     CALL DSKCON

        LDA     6,X         GET ERROR CODE
        BNE     ERROR       BRANCH ON ERROR
```

J & M SYSTEMS, LTD.

Write Track (Opcode Ø4)

This operation is intended only for disk formatting. It is recommended that this operation not be used unless absolutely necessary since it can be quite destructive! For more information, refer to the Western Digital FD-179X-Ø2 Floppy Disk Controller Data Sheet.

Select Drive (Opcode Ø5)

The Select Drive Operation may be used to select a drive. All other operations will automatically select the requested drive, so it should not normally be necessary to use the Select Drive function. However, if required, the Select Drive Operation may be used to select a drive without performing any other function.

Example:

In the following example, Drive 2 is selected:

```
SELECT    LDX     #$CØØ6      POINT TO DPB
          LDA     #Ø5         SELECT DRIVE OP CODE
          STA     ,X
          LDA     #Ø2         SELECT DRIVE 2
          STA     1,X
          JSR     [$CØØ4]     CALL DSKCON
```

Home Head (Opcode Ø6)

The Home Head Operation (Opcode Ø6) is identical in every respect to Opcode ØØ.

Home Head (Opcode Ø7)

The Home Head Operation (Opcode Ø7) is identical in every respect to Opcode ØØ.

Miscellaneous Notes

1. DSKCON preserves the contents of all registers on the hardware stack (Register "S"), so all registers are undisturbed on return.

2. DSKCON controls the drive motor shut-off function via an IRQ Interrupt. So, if DSKCON is to control motor shut-off, the IRQ must be enabled before DSKCON is called. The drive motors may be turned off at any time by writing a ØØ to memory location FF4Ø Hex.

3. DSKCON does not change any of the paramters in the DPB with the exception of FDCSTA. So, it is not necessary to set all of the DPB parameters on each call to DSKCON. Only the parameters which need to be changed from the last operation must be set.

APPENDIX C

ERROR CODES AND MESSAGES

AE Already Exists

PROBLEM: File name specified as a new file name in a RENAME Command already exists in the directory.

CORRECTIVE ACTION: Use a different file name or delete the existing file with the same name.

AO Already Open

PROBLEM: An attempt was made to open a file which was already open.

CORRECTIVE ACTION: Close the file. A file must be closed before it can be opened.

BR Bad Record Number

PROBLEM: The record specified does not exist in a direct-access file.

CORRECTIVE ACTION: Use the LOF Function to determine the number of records in the file. Do not use a record number greater than that returned by the LOF Function.

BM Bad Memory

PROBLEM: Occurs when LOADM or RAM Command attempts to load non-existent memory or Read-Only Memory (ROM).

CORRECTIVE ACTION: Do not use the RAM Command if your computer has less than 64k bytes of RAM. In the case of a LOADM Command, either the [offset] parameter is too large, or the <execution> address parameter used when the program was saved was incorrect.

BI Backup Incompatibility

PROBLEM: Occurs in response to the BACKUP Command when the source disk and the destination disk are formatted differently.

CORRECTIVE ACTION: Use the DSKINI Command to format the destination disk to the same format as the source disk. That is, if the source disk is 2-sided, 40-track, then the destination disk must also be formatted 2-sided, 40 track.

DF Disk Full

PROBLEM: There is not enough unallocated disk space remaining on the disk to complete the command.

CORRECTIVE ACTION: Either delete one or more files on the disk or use another disk with more unallocated space.

DN Drive Number

PROBLEM: A drive number has been specified which is not allowed.

CORRECTIVE ACTION: Use another drive number. Only drive numbers of 0, 1, or 2 are allowed.

ER End of Record

PROBLEM: In a direct-access file, an attempt has been made to transfer data beyond the end of the record. The length of the record specified with the OPEN Statement must not be exceeded.

CORRECTIVE ACTION: Check the record length specification in the OPEN Statement. If the record length specification is correct, do not attempt to transfer more data than may be contained in a record of the length specified.

FN File Name Error

PROBLEM: A file name has been specified which includes characters that are not allowed, or more than the maximum allowed number of characters.

CORRECTIVE ACTION: Correct the file name. A file name may contain any character except a slash ("/"), and must be no more than 8 characters in length.

FD File Data Error

PROBLEM: An attempt has been made to transfer data between a file buffer and a variable of a different type. For example, if the data in the buffer is string data, and the variable is numeric, this error will occur.

CORRECTIVE ACTION: Use string variables to transfer string data fields, and numeric variables for numeric fields.

FM File Mode Error

PROBLEM: An attempt has been made to access a file in a mode other than that for which it was opened. For example, an attempt to write data to a sequential access file which is open for input ("I" Mode), will cause an FM Error.

CORRECTIVE ACTION: Either close the file and re-open it for the correct mode, or change the statement in error to agree with the mode for which the file was opened.

FO Field Overflow Error

PROBLEM: An attempt has been made to transfer more data to a field variable than the variable is defined to contain. For example, if a variable has been FIELDed as a 5-byte field, and an attempt is made to transfer more than 5 bytes of data to it, an FO Error will occur.

CORRECTIVE ACTION: Either re-define the FIELD variable, or re-define the data being transferred to it.

FS File Structure Error

PROBLEM: The disk granule table in the directory has been altered in such a way that the granules do not correspond correctly with the files in the directory.

CORRECTIVE ACTION: The disk must be re-formatted. Use the COPY Command to save as many files as possible, then re-format the disk.

IE Input past end of file

PROBLEM: An attempt has been made to read more data from a file than is contained in the file.

CORRECTIVE ACTION: Use the EOF and/or LOF Functions to determine the file length. Do not attempt to read more data from the file than it contains.

IO Input/Output Error

PROBLEM: An error has occured during disk I/O. The error may be due to a number of problems such as CRC Errors, no disk in drive, etc. The exact cause of the problem may be determined by using the command PRINT PEEK (&HØØFØ).

This command will return an error code number in the range of ØØ-255. The error code is a bit-mapped code. Refer to the description of the "Completion Code Parameter" in Appendix B for a complete description of the interpretation of this code.

NE Name Error

PROBLEM: The file name specified in a command was not found in the disk directory. The file does not exist.

CORRECTIVE ACTION: Either the file name was incorrectly specified, or the file may exist on another disk.

NO Not Open Error

PROBLEM: An attempt has been made to transfer data to or from a disk file which has not been opened with an OPEN Statement.

CORRECTIVE ACTION: Do not attempt to read or write to or from a file unless the file has been opened with an OPEN Statement.

OB Out of Buffer Space Error

PROBLEM: Insufficient memory space has been allocated to the Direct-access record buffer.

CORRECTIVE ACTION: Use the FILES Statement to allocate more buffer space.

OM Out of Memory Error

PROBLEM: All of the memory in the computer has been filled. An attempt has been made to load more memory than is available.

CORRECTIVE ACTION: Reset the computer and try again. If you do not have a 64k byte system, you may have to expand the memory before you can load some files.

OS Out of String Space

PROBLEM: Insufficient memory space has been allocated to the storage of string variables.

CORRECTIVE ACTION: Use the CLEAR Command to allocate more string space.

SE Set Error

PROBLEM: An attempt has been made to RSET or LSET data to a non-fielded variable. These functions may be used only with fielded variables.

CORRECTIVE ACTION: Use the RSET and LSET Functions only with variables which have been defined in a FIELD Statement.

SR Step-rate Error

PROBLEM: An attempt has been made to select a head step rate which is not allowed.

CORRECTIVE ACTION: When using the RATE Command, only step rate codes in the range of Ø-3 are allowed.

SN Syntax Error

PROBLEM: A command or statement has been encountered which is unrecognized.

CORRECTIVE ACTION: Re-enter the command or statement. Check the spelling and punctuation for correct syntax.

ST String Error

PROBLEM: A string operation has been specified which is too complex to be processed.

CORRECTIVE ACTION: Break the operation up into smaller, less complex steps on separate lines.

TM Type Mismatch

PROBLEM: The data type of a variable does not correspond to the data which is being assigned to it. For example, if an attempt is made to assign numeric data to a string variable, the TM Error will occur (eg, A$=5 or A="ABC")

CORRECTIVE ACTION: Do not attempt to assign numeric data to a string variable, or to assign string data to a numeric variable.

UL Undefined Line Error

PROBLEM: An attempt has been made to LIST, GOTO, or GOSUB to a line number which does not exist.

CORRECTIVE ACTION: Correct the specified line number, or enter the line in question.

VF Verification Error

PROBLEM: This error occurs during a disk write operation when the VERIFY Option is ON, and the data read from the disk does not match that which was written.

CORRECTIVE ACTION: Re-issue the command. Many VF Errors will not persist. If the problem persists, you may need to re-format the disk or use a new disk.

WP Write Protect Error

PROBLEM: An attempt has been made to write to a disk which is write-protected.

CORRECTIVE ACTION: Either remove the write-protect tab, or use another disk.

ERROR CODE SUMMARY

CODE	MEANING
AE	Already Exists
AO	Already Open
BR	Bad Record number
BM	Bad Memory
BI	Backup Incompatibility
DF	Disk Full
DN	Drive Number
ER	End of Record
FN	File Name error
FD	File Data error
FM	File Mode error
FO	Field Overflow
FS	File Structure error
IE	Input past End of file
IO	Input/Output error
NE	Name Error
NO	Not Open
OB	Out of Buffer space
OM	Out of Memory
OS	Out of String space
SE	Set Error
SR	Step Rate error
SN	SyNtax error
ST	STring error
TM	Type Mismatch error
UL	Undefined Line
VF	VeriFication error
WP	Write Protect error

This page intentionally blank

APPENDIX D

INSTALLATION OF THE J&M SYSTEMS DISK CONTROLLER

To install the J&M Systems "COCO" Disk Controller, carefully follow the step-by-step instructions outlined below:

STEP 1: TURN OFF ALL POWER! This is most important. Never connect or disconnect the controller with power on! Be sure that all power is off - Disk drives, computer, everything! Connecting or disconnecting the controller with power on may destroy the controller, the computer, or the disk drive(s)!

STEP 2: Locate the ROM Cartridge slot on the right-hand side of the computer, near the rear. Insert the controller into this slot carefully, but firmly. Orient the controller such that the J&M Systmes logo is up and the arrows along the edge are pointing in to the cartridge slot. Be sure that the controller is completely seated in the slot, and that it is not crooked (see Figure A and B).

STEP 3: Connect the ribbon cable from the controller to the disk drive or drives. A polarity mark on the cable connector marks Pin 1. Pin 1 of the controller connector is on the upper side of the connector, toward the edge opposite the J&M Logo (see Figure C). Be sure that the cable is firmly seated on the connector.

STEP 4: Plug the computer and the disk drives into a SINGLE power outlet or a SINGLE power strip. This is to insure that the computer and drives have a common ground. FAILURE TO PROVIDE A COMMON GROUND CAN DESTROY COMPUTER, CONTROLLER AND DRIVES.

STEP 5: Make sure that the disk drive door(s) are open, then turn on all power to the computer, monitor, disk drive(s), etc. Never turn power on or off with the disk drive doors closed, especially when there is a disk in the drive!

STEP 6: You should now see the message below on the monitor. The system is ready to go. Happy computing!

```
JDOS Disk Basic Operating System
Copyright 1983 J&M Systems, Ltd.
OK
```

figure A

September 1984

15100A Central Ave. SE • Albuquerque, New Mexico 87123 • (505)292-4182

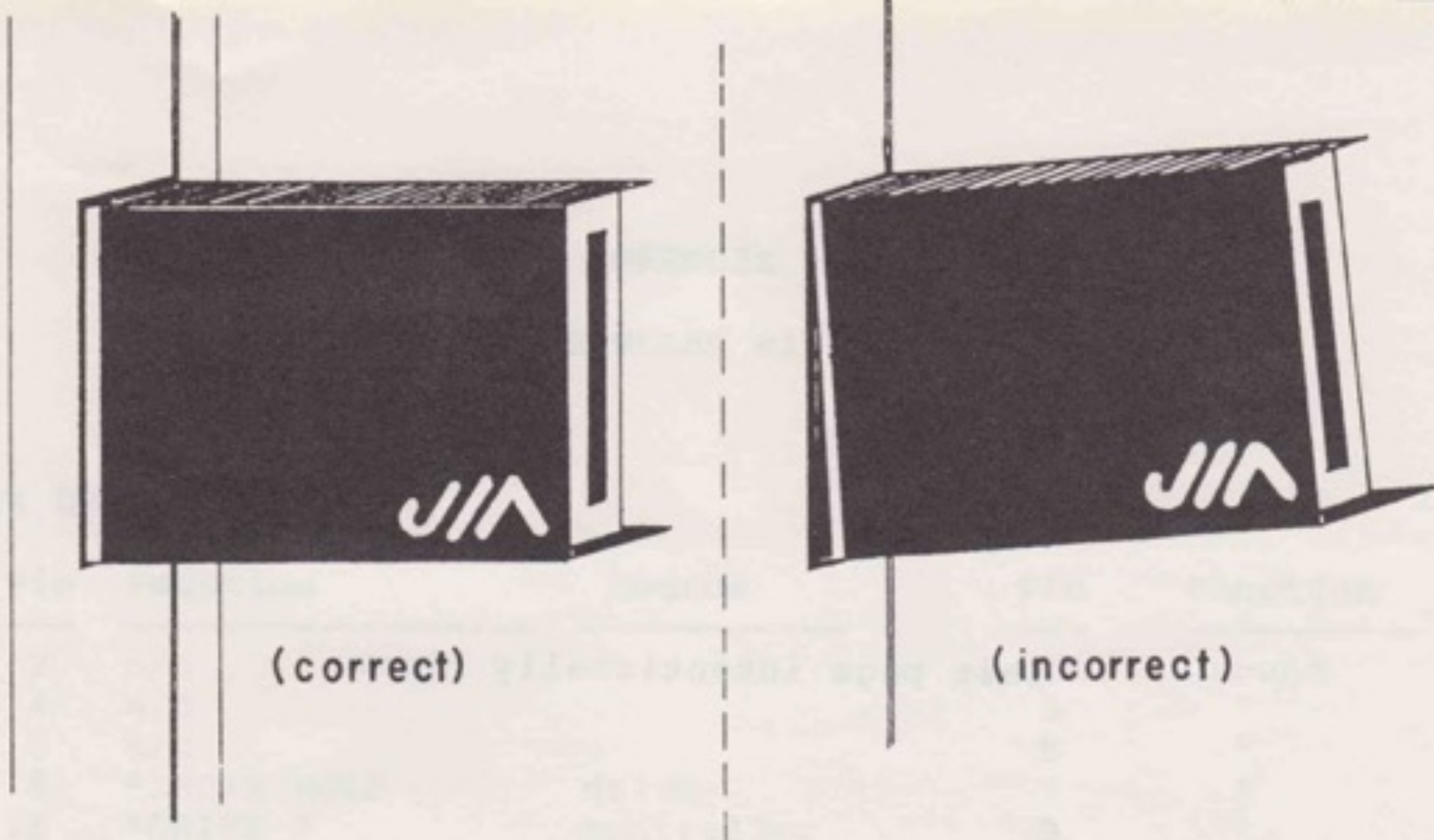

figure B

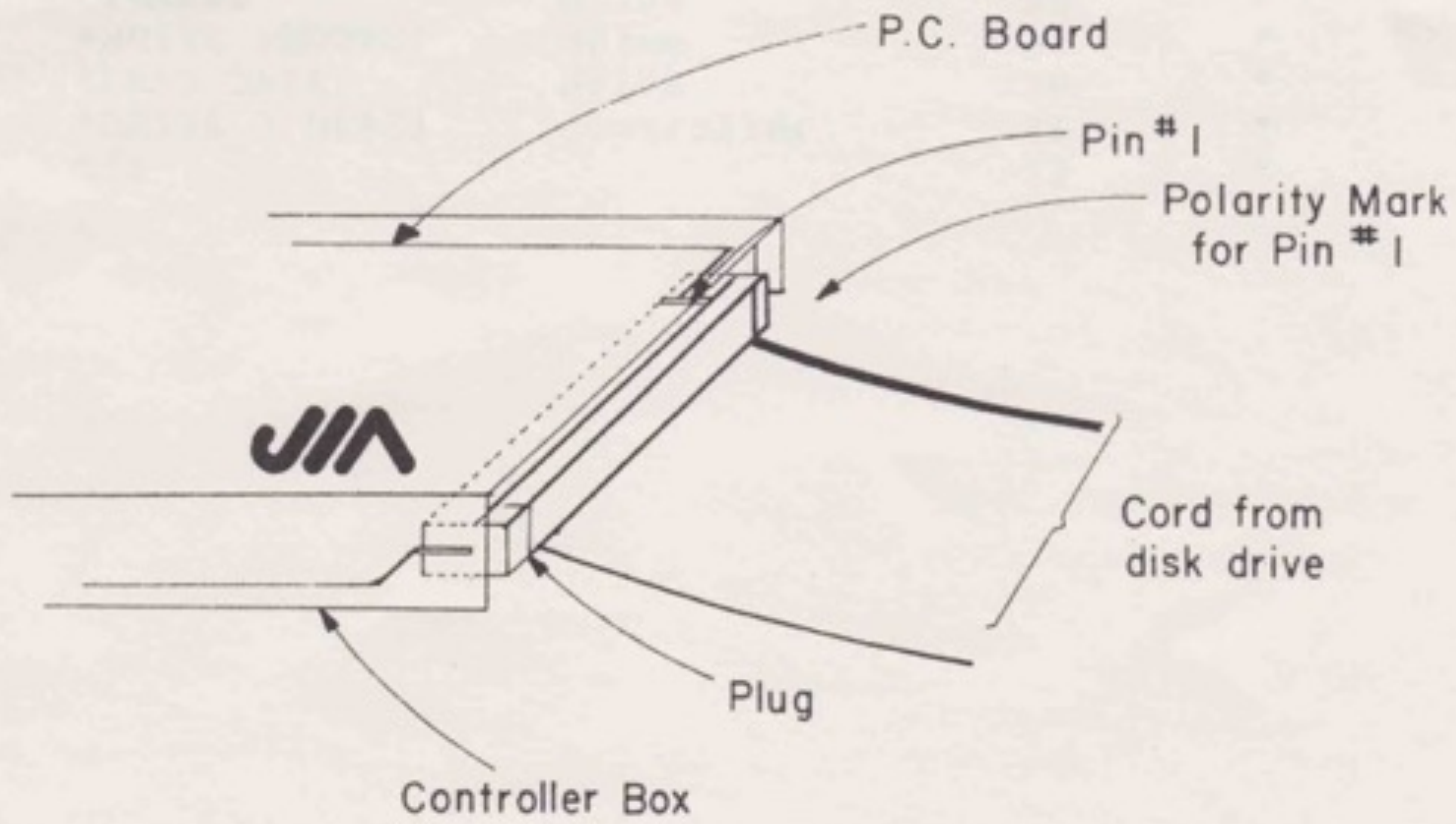

figure C

J & M SYSTEMS, LTD.

15100A Central Ave. SE • Albuquerque, New Mexico 87123 • (505)292-4182

This page intentionally blank

APPENDIX E

CONTROLLER PINOUTS

DISK DRIVE CONNECTOR

Pin	Function	Source	Pin	Function
2	n/c		1	Ground
4	n/c		3	"
6	n/c		5	"
8	*INDEX HOLE	drive	7	"
10	*DRIVE_0	controller	9	"
12	*DRIVE_1	controller	11	"
14	*DRIVE_2	controller	13	"
16	*MOTOR_ON	controller	15	"
18	*DIRECTION	controller	17	"
20	*STEP	controller	19	"
22	*WRITE_DATA	controller	21	"
24	*WRITE_GATE	controller	23	"
26	*TRACK0	drive	25	"
28	*WRITE_PROTECT	drive	27	"
30	*READ_DATA	drive	29	"
32	*DRIVE_3/HEAD1	controller	31	"
34	n/c		33	"

COLOR COMPUTER CONNECTOR

Pin	Function	Source
1		
2		
3	*HALT	Controller
4	*NMI	Controller
5	*RESET	Computer
6	E-CLOCK	Computer
7		
8		
9	+5 Volts	Computer
10	D0	Bi-Directional
11	D1	"
12	D2	"
13	D3	"
14	D4	"
15	D5	"
16	D6	"
17	D7	"
18	R/*W	Computer
19	A0	"
20	A1	"
21	A2	"
22	A3	"
23	A4	"
24	A5	"
25	A6	"
26	A7	"
27	A8	"
28	A9	"
29	A10	"
30	A11	"
31	A12	"
32	*CTS	"
33	Ground	
34	Ground	
35		
36	SCS*	Computer

APPENDIX F

INTERFACING MACHINE-LANGUAGE PROGRAMS TO JDOS

The release of Radio Shack DOS 1.1 and J & M System's JDOS has brought about a minor crisis within the Color Computer Community. Many Color Computer users have found that much of their existing commercial software will not run under these new Operating Systems. All too often, the new Operating Systems have been blamed for this problem when, in fact, the problem has been caused by commercial software which makes use of undocumented subroutines in the Radio Shack DOS 1.Ø.

What do we mean by "making use of undocumented subroutines"? We are referring to the use of subroutines within the Color Computer ROM's which are not documented by the original manufacturer. Unfortunately, there has come to be a school of thought within the computer industry that the only reason for not documenting system software source code is for copyright security. But there are other valid reasons. One reason, which is so often overlooked, is the right of freedom to change.

When a subroutine is documented, then there is at least an implied commitment to maintain that subroutine throughout all revisions of the product. But, there is no obligation on the part of the vendor, implied or otherwise, to maintain non-documented subroutines in future releases of the product.

If there is anything in this rapidly-advancing computer industry which is constant, it is change! Change occurs for many reasons. Advances in technology, upgrading of capability, and correction of defects or errors, to name a few. The point is, change does occur, and it occurs often. When a vendor releases a new software system, it is only reasonable to assume that changes to that system will occur. And this is especially true when the system is well accepted within the industry.

And so it was with Radio Shack DOS 1.Ø. And so it should be! Any vendor must retain the right to change their product as they see fit, for whatever reason, if that product is to survive in the market place! This brings us to the problem at hand.

Shortly after the release of Radio Shack DOS 1.Ø, there was a flurry of articles and books offering the "secret" of the Radio Shack software in the form of commented, disassembled listings. And, indeed, there is a valid use for this type of material. It provides the end user with a better understanding of the software and hardware that they have purchased. And, it provides examples to those of us who are interested in learning new techniques in programming and reaping the maximum that our computer has to offer. But, using this information to design non-documented interfaces to applications software can create problems!

The only subroutine within the Radio Shack DOS 1.Ø which was documented by Radio Shack was (and is) the "DSKCON" Subroutine which provides access to the disk drivers. Apparently, this was the only subroutine that Radio Shack was willing to maintain. Since none of the other routines within the ROM were documented by Radio Shack, they certainly felt no obligation in maintaining interface compatibility when they changed the ROM. Note that they did maintain compatibility with the DSKCON Routine!

When J & M Systems designed JDOS, there was no compatibility whatsoever at this level. JDOS is, of course, completely different than Radio Shack DOS internally! And when revisions are made to JDOS, there is no guarantee that any of the internal subroutines will remain unchanged.

Apparently, a number of software authors who created software to run on the Color Computer made use of undocumented subroutines within the Color Computer ROM's. Why? Perhaps they were in a hurry to get their products to market. Or, maybe it simply didn't occur to them that the ROM might be changed sometime in the future. But in any case, the inevitable has happened. The ROM's have been changed, and suddenly we find our programs making calls to subroutines which no longer exist, or have been relocated or modified in such a way that they no longer perform the function for which they were originally used.

If the Color Computer is to continue to evolve and keep pace with the industry, then there must be freedom to change at the system software level. The market place will not evolve if independent software vendors must constantly re-write existing software to keep pace with new revisions of system software! To that end, we offer the following solution.

In order to understand why the technique described here is preferable, some discussion is in order. It is important to realize that the Color Computer ROM's are a BASIC Interpreter, and as such, must guarantee compatibility through all revisions with any program written in BASIC. However, even though the interpreter does provide disk I/O functions, it was not intended to provide support at the machine code level.

When writing machine code programs, it is not necessary to use the ROM Routines at all. However, there is much capability there which need not be duplicated if it is used properly. The machine language programmer can best take advantage of this capability by thinking of the ROMs as a BASIC Interpreter and interfacing to them on that level. This will guarantee compatibility through all ROM revisions, since the ROM's themselves must maintain compatibility at this level.

The technique presented here is really quite straightforward. The basic concept is to interface to the BASIC Interpreter in the same manner as commands which are entered from the keyboard. One restriction must be noted. Lower memory (0000-1600 Hex) must be left intact. This area of memory provides the interface between the machine-code program and the BASIC Interpreter, as well as a work-area for the interpreter. This memory may not be used by the machine-language program except as described here.

During initialization, the BASIC Interpreter will construct three tables in lower memory which include the following data:

Table #1 for BASIC

ADDRESSES (hex)	DESCRIPTION
0120	Number of command reserved words
0121-0122	Table of reserved command words
0123-0124	Jump addresses for interpretation and execution of BASIC Commands
0125	Number of Basic functions
0126-0127	Table of reserved function names
0128-0129	Jump addresses for interpretation and execution of Basic functions

Table #2 for Extended BASIC

ADDRESSES (hex)	DESCRIBTION
012A	Number of command reserved words
012B-012C	Table of reserved command words
012D-012E	Jump addresses for interpretation and execution of a Extended commands
012F	Number of Extended functions
0130-0131	Table of reserved function names
0132-0133	Jump addresses for intrpretation and execution of Extended functions

Table #3 for J & M System's JDOS and Radio Shack's DOS

ADDRESSES (hex)	DESCRIBTION
0134	Number of command reserved words
0135-0136	Table of reserved command words
0137-0138	Jump addresses for interpretation and execution of disk commands
0139	Number of Disk functions
013A-013B	Table of reserved function names
013C-013D	Jump addresses for interpretation and execution of disk functions

These tables are used by the interpreter during command interpretation and execution. After a command is entered, the interpreter attempts to match the command with an entry in the command lists in each of the ROM's. If a match is found, a one-byte representation of the command, called a "token", is returned. The token is then passed to interpretation and execution routines in the ROM's.

Each ROM (BASIC, Extended BASIC, and Disk BASIC) compares the token passed to those that correspond to its own commmands and functions. When the token is matched, the appropriate routine is called to execute the command. The command is executed and control is passed either to an error handling routine or back to the calling routine.

This same basic process may be invoked by a machine-code program. However, there are some short-cuts that may be used to simplify the process. First, to simulate command input, the program must set up the required information in a characacter string and set a pointer for the interpreter (see example). Secondly, the matching of tokens does not need to be performed. The token (see table of token values) only needs to be passed to the interpretation and execution routine. Lastly, the program should use a JSR instruction to "call" the interpreter so that control will be returned back to the machine-code program.

The final consideration of this technique is that of error trapping. Error trapping within the interpreter is accomplished by changing addresses in a jump-table. This jump-table is initialized after power-up and contains addresses for routines that are dependent upon the specific ROM's that the Color Computer contains. For example, if the Extended BASIC ROM is present, the addresses in the jump-table will correspond with Extended BASIC software.

The only address within the jump-table that is of interest here is the error routine vector address which resides at Ø191 Hex. The vector at this address should be set to point to an error routine within the machine-code program. Then, if an error is encountered during execution of a BASIC command supplied by the machine code program, the routine encountering the error will always pass control back to the machine-code program.

A few examples will illustrate the procedure:

Example 1: This example will load and run a binary file.

```
        ORG  $28ØØ
PROGNM  FCC  '"MYPROG.BIN:1"'     NEEDS TO BE FOLLOWED BY A
        FCB  $Ø                   NULL BYTE
START   LDA  #$D3                 TOKEN OF LOADM COMMMAND
        LDX  #PROGNM              STORE ADDRESS OF STRING
                                  FOR EVALUATION
        STX  #A5                  POINTER IN OPERATING
                                  SYSTEM FOR INTREPRATATION
        JSR  [$Ø137]              EXECUTE DISK COMMAND
        LDA  #$A2                 TOKEN FOR EXEC COMMAND
        JSR  [$Ø123]              EXECUTE COMMAND (IF NO
                                  RETURN IS NECASSARY USE
                                  THE JMP DIRECTIVE)
        END
```

Example 2: This example demonstrates use of the RENAME Command.

```
        ORG  $28ØØ
  NAMES FCC  '"MYPROG/BAS:1","YOURPROG:Ø"'
        FCB  $Ø                   NEEDS A NULL BYTE
  START LDA  #$D6                 TOKEN FOR RENAME
        LDX  #NAMES               STRING FOR EVALUATION
        STX  $A5                  INTERPRETATION POINTER
        JSR  [$Ø137]              EXECUTE DISK COMMAND
        END
```

Example 3: This example uses numbers in a command.

```
        ORG  $28ØØ
  COMAD FCC  '"Ø,17,3,"THIS IS A TEST","THIS IS SECOND"'
        FCB  $Ø                   REQUIRES NULL BYTE
  START LDA  #$EØ                 TOKEN FOR DSKO$
        LDX  #COMAD               STRING FOR EVALUATION
        STX  $A5                  INTERPRETATION POINTER
        JSR  [$Ø137]              EXECUTE THE COMMAND
        END
```

As can be seen from these examples, this technique is very simple and straight-forward to use. Furthermore, programs adhering to this technique will be compatible with any future releases or updates of the ROM's in the Color Computer.

A Table of Token values for commands and functions follows:

BASIC:

COMMAND	**TOKEN (hex)**	**COMMAND**	**TOKEN (hex)**
FOR	80	STOP	91
GO	81	POKE	92
REM	82	CONT	93
REM	83	LIST	94
ELSE	84	CLEAR	95
IF	85	NEW	96
DATA	86	CLOAD	97
PRINT	87	CSAVE	98
ON	88	OPEN	99
INPUT	89	CLOSE	9A
END	8A	LLIST	9B
NEXT	8B	SET	9C
DIM	8C	RESET	9D
READ	8D	CLS	9E
RUN	8E	MOTOR	9F
RESTORE	8F	AUDIO	A1
RETURN	90	EXEC	A2
		SKIPF	A3

FUNCTION	**TOKEN (hex)**	**FUNCTION**	**TOKEN (hex)**
SGN	80	ASC	8A
INT	81	CHR$	8B
ABS	82	EOF	8C
USR	83	JOYSTK	8D
RND	84	LEFT$	8E
SIN	85	RIGHT$	8F
PEEK	86	MID$	90
LEN	87	POINT	91
STR$	88	INKEY$	92
VAL	89	MEM	93

NOTE: All functions must be preceeded with FF Hex.

EXTENDED BASIC:

COMMANDS	TOKEN (hex)	COMMANDS	TOKEN (hex)
DEL	B5	COLOR	C1
EDIT	B6	CIRCLE	C2
TRON	B7	PAINT	C3
TROFF	B8	GET	C4
DEF	B9	PUT	C5
LET	BA	DRAW	C6
LINE	BB	PCOPY	C7
PCLS	BC	PMODE	C8
PSET	BD	PLAY	C9
PRESENT	BE	DLOAD	CA
SCREEN	BF	RENUM	CB
PCLEAR	CØ	FN	CC
		USING	CD

FUNCTION	TOKEN (hex)	FUNCTION	TOKEN (hex)
ATN	94	SQR	9B
COS	95	HEX$	9C
TAN	96	VARPTR	9D
EXP	97	INSTR	9E
FIX	98	TIMER	9F
LOG	99	PPOINT	AØ
POS	9A	STRING$	A1

DISK BASIC:

COMMAND	TOKEN (hex)	COMMAND	TOKEN (hex)
DIR	CE	RSET	D7
DRIVE	CF	SAVE	D8
FIELD	DØ	WRITE	D9
FILES	D1	VERIFY	DA
KILL	D2	UNLOAD	DB
LOAD	D3	DSKINI	DC
LSET	D4	BACKUP	DD
MERGE	D5	COPY	DE
RENAME	D6	DSKI$	DF
		DSKO$	EØ

EXTRA JDOS COMMANDS

AUTO	E1
DOS	E2
ERROR	E3
RATE	E4
FLEX	E5
..	E6
RAM	E7

FUNCTION	TOKEN (hex)	FUNCTION	TOKEN (hex)
CVN	A2	LOF	A5
FREE	A3	MKN	A6
LOC	A4	AS	A7

EXTRA JDOS FUNCTIONS

ERL	A8
ERR	A9

J & M SYSTEMS, LTD.