

NEWDOS/80

FOR THE TRS-80

MODEL I / III / 4

MICRO COMPUTER

Apparat Incorporated takes pleasure in presenting NEWDOS/80, Version 2.5. Above is the registration number of your NEWDOS/80. This registration number must be the same as the registration number you find on your diskette label and the enclosed registration card. If they are not, return them to the dealer from whom you purchased your NEWDOS/80 to be reissued. This registration Number is your assurance of receiving any corrections or minor revisions to NEWDOS/80 that may be released. The registration card should be completed and returned to Apparat at your earliest convenience. **PLEASE RETURN THE CARD IT IS IMPORTANT!** It our only method of determining who has purchased this copy of the system. This number should be included in all correspondence with Apparat.



Apparat, Inc

4401 So. Tamarac Parkway • Denver, Colorado 80237

HELPFUL HINTS IN USING YOUR NEWDOS/80 VERSION 2.5

We suggest using the following checklist as a guide to setting up software on your Hard Disk System:

1. Carefully read through all related documentation.
2. Make hardware installation as directed by instructions supplied with your Hard Drive Unit.
3. Boot on your NEWDOS/80 Version 2.5 original master diskette and make backup copy or copies. Refer to Chapter 1, Section 1.4 of the NEWDOS/80 Version 2.0 manual for details if unfamiliar with the procedure.
4. Designate one of your backups as a working copy and boot (or reset) on it. Use this diskette for the remainder of this procedure.
5. Use HDFMTAPP/CMD. Refer to Section 4 in the NEWDOS/80 Version 2.5 manual (Appendix C).
6. Set up your PDRIVE definitions for the Hard Drive Volumes according to Section 3 & 6 of the NEWDOS/80 Version 2.5 manual.
7. Use DOS command FORMAT on each defined volume of the Hard Disk. This has not been very clear to some of our users. HDFMTAPP/CMD is only a "media format" utility that allows the hard drive to be used with NEWDOS/80 (or other DOS system). DOS command FORMAT, although does not prepare the media as HDFMTAPP/CMD, is important for checking the media and preparing the directory (DIR/SYS) and the boot sectors (BOOT/SYS) on the hard drive volumes. If the volume was not prepared with FORMAT, it will seem to operate correctly with COPY, OPEN, and CLOSE (ie. normal file operations), but HDBACKUP/CMD and DIRCHECK/CMD will not function.
8. (Optional) Move NEWDOS/80 Version 2.5 to Hard Disk Volume. Refer to Section 5 of the NEWDOS/80 Version 2.5 manual. Don't forget to move your PDRIVE definitions to appropriate new slots. Follow procedure to create Boot Diskette.
9. Install application software and related data files.

Table of Contents

Chapter 1. INTRODUCTION

1.1.	Registration.....	1-1
1.2.	Trademark Credits.....	1-1
1.3.	What Is Apparat's DOS/80 Version 2?.....	1-1
1.4.	Duplicate and Specify the System.....	1-2
1.5.	Apply Outstanding Zaps.....	1-4
1.6.	Commence Using NEWDOS/80.....	1-5
1.7.	Apparat Thanks Its Beta Testers.....	1-5

Chapter 2. DOS LIBRARY COMMANDS

2.1.	Notation Conventions and General Information.....	2-1
2.2.	APPEND Append one file onto the end of another.....	2-2
2.3.	ATTRIB Assign attributes to a file.....	2-3
2.4.	AUTO Define the DOS command to be executed at reset.....	2-5
2.5.	BASIC2 Activate non-disk BASIC (Model I only).....	2-5
2.6.	BLINK Enable/disable cursor blinking.....	2-5
2.7.	BOOT Reset the computer.....	2-6
2.8.	BREAK Enable/disable the BREAK key.....	2-6
2.9.	CHAIN Shift to keyboard input from disk.....	2-6
2.10.	CHNON Alter chaining state.....	2-7
2.11.	CLEAR Clear user memory routes, timer and logical enqueues.....	2-8
2.12.	CLOCK Display the time every second.....	2-9
2.13.	CLS Clear the display.....	2-9
2.14.	COPY Copy single or multiple files or a full diskette.....	2-9
2.15.	CREATE Pre-allocate a disk file.....	2-18
2.16.	DATE Set computer's current date.....	2-19
2.17.	DEBUG enable or disable the DEBUG facility.....	2-20
2.18.	DIR Display a diskette's directory information.....	2-20
2.19.	DO Shift to keyboard input from disk.....	2-22
2.20.	DUMP Dump memory contents to disk.....	2-22
2.21.	ERROR Display DOS error message.....	2-24
2.22.	FORMAT Format a diskette for use with the NEWDOS/80 system.....	2-24
2.23.	FORMS (Model III only) Set printer parameters.....	2-26
2.24.	FREE Display free granule count of each mounted diskette.....	2-27
2.25.	HIMEM Set DOS's high memory value.....	2-27
2.26.	JKL Send current contents of display to the printer.....	2-27
2.27.	KILL delete a file.....	2-28
2.28.	LC Set keyboard a-z toggle switch to specified state.....	2-29
2.29.	LCDVR (Model I only) Lower case driver.....	2-29
2.30.	LIB Display NEWDOS/80 library commands.....	2-30
2.31.	LIST List a text file on the display.....	2-30
2.32.	LOAD Load a Z-80 machine language file into RAM.....	2-31
2.33.	MDBORT Terminate MINI-DOS and go to DOS READY.....	2-31
2.34.	MDCOPY Copy a file while under MINI-DOS.....	2-32
2.35.	MDRET Exit from MINI-DOS and return to main program.....	2-32
2.36.	PAUSE Display message and pause waiting on ENTER.....	2-33
2.37.	PDRIVE Assign default attributes to a physical drive.....	2-33
2.38.	PRINT List a text file on the printer.....	2-39
2.39.	PROT Alter some diskette control data.....	2-40

2.40.	PURGE	Selectively kill files from a diskette.	2-41
2.41.	R	Repeat the previous DOS command.	2-41
2.42.	RENAME	Rename a file.	2-42
2.43.	ROUTE	Route one device to or from another	2-42
2.44.	SETCOM	(Model III only) Set RS-232 interface parameters.	2-44
2.45.	STMT	Display specified message.	2-45
2.46.	SYSTEM	Change system options.	2-45
2.47.	TIME	Set the real time clock.	2-50
2.48.	VERIFY	Require verify read after every disk write.	2-51
2.49.	WRDIP	Write directory sectors protected.	2-52

Chapter 3. DOS ROUTINES

3.1.	Specifications Defined	3-1
3.2.	402DH No-Error Exit	3-1
3.3.	4030H Error-already-displayed DOS Error Exit	3-2
3.4.	4400H No-Error Exit. Performs identical to 402DH.	3-2
3.5.	4405H Enter DOS and execute a command	3-2
3.6.	4409H DOS Error Exit	3-2
3.7.	440DH Enter DEBUG	3-3
3.8.	4410H Enqueue a user timer interrupt routine.	3-3
3.9.	4413H Dequeue a user timer interrupt routine.	3-4
3.10.	4416H Keep drives rotating	3-4
3.11.	4419H DOS-CALL Execute a DOS command and return.	3-4
3.12.	441CH Extract a filespec	3-5
3.13.	4420H Open a FCB to a new or existing disk file	3-5
3.14.	4424H OPEN a FCB to an existing file	3-6
3.15.	4428H CLOSE a FCB. Conditions 3.1.A, B and C hold	3-7
3.16.	442CH Kill the FCB's associated file	3-7
3.17.	4430H Load a program file	3-7
3.18.	4433H Load and commence execution of a program file	3-7
3.19.	4436H Read sector or logical record from disk	3-7
3.20.	4439H Write sector or logical record to disk	3-8
3.21.	443CH Write sector or logical record to disk with verify ...	3-9
3.22.	443FH Position FCB to start of file	3-9
3.23.	4442H Position FCB to a specified file record	3-9
3.24.	4445H Position FCB back one record	3-9
3.25.	4448H Position FCB to EOF	3-9
3.26.	444BH Allocate file space	3-10
3.27.	444EH Position FCB to the specified RBA	3-10
3.28.	4451H Write the EOF value from the FCB to the directory	3-10
3.29.	445BH Select and power up the specified drive	3-10
3.30.	445EH Test for mounted diskette	3-10
3.31.	4461H *Name routine enqueue	3-10
3.32.	4464H *name routine dequeue	3-11
3.33.	4467H Send message to the display	3-11
3.34.	446AH Send message to the printer	3-11
3.35.	446DH Convert clock time to HH:MM:SS character format	3-11
3.36.	4470H Convert the date to MM/DD/YY character format	3-11
3.37.	4473H Insert default name extension into filespec	3-12
3.38.	0013H Read a byte from a disk file	3-12
3.39.	001BH Write a byte to a disk file	3-12
3.40.	447BH Model III only (performs as Model I 4410H)	3-12

Chapert 4. DOS FEATURES

4.1.	DEBUG Facility	4-1
4.2.	MINI-DOS	4-5
4.3.	CHAINING	4-7
4.4.	DOS-CALL	4-12
4.5.	JKL	4-13
4.6.	Asynchronous Execution	4-14

Chapter 5. DOS MODULES, DATA STRUCTURES, AND MISCELLANEOUS INFORMATION

5.1.	Files required on each diskette used with NEWDOS/80	5-1
5.2.	NEWDOS/80 DOS System Modules	5-1
5.3.	NEWDOS/80 BASIC Modules	5-2
5.4.	Other Modules on the NEWDOS/80 diskette	5-3
5.5.	Reduced Sized System.	5-4
5.6.	Diskette Directory Structure	5-4
5.7.	FPDE File Primary Directory Entry	5-7
5.8.	FXDE File Extended Directory Entry	5-9
5.9.	FCB File Control Block	5-9

Chapter 6. ADDITIONAL PROGRAMS SUPPLIED ON NEWDOS/80 DISKETTE

6.1.	SUPERZAP Inspect/Change Disk/Main Memory	6-1
6.2.	DISASSEM Disassemble Z-80 Code	6-5
6.3.	LMOFFSET Move Module to New Load Position	6-9
6.4.	DIRCHECK Inspect and List a Directory	6-12
6.5.	EDTASM Disk Oriented Editor/Assembler	6-14
6.6.	CRAINBLD Create and Modify Chain Files	6-16
6.7.	ASPOOL Automatic Spooler	6-19

Chapter 7. DISK BASIC, NON-I/O ENHANCEMENTS

7.1.	INTRODUCTION, Requirements	7-1
7.2.	General comments	7-1
7.3.	Activating DISK BASIC	7-2
7.4.	Direct Scrolling/Editing Commands	7-3
7.5.	Text Editing Command Truncation	7-4
7.6.	DI and DU text editing functions	7-4
7.7.	RUN and LOAD (optionally retaining variables)	7-4
7.8.	MERGE Dynamic loading of overlay program	7-5
7.9.	RENUM Renumber the Current BASIC Program.	7-5
7.10.	REF List references to variables, line numbers and keywords	7-7
7.11.	Lower Case Suppression (Model I only)	7-8
7.12.	RUN-ONLY	7-8
7.13.	Comarisons in the use of CMD between NEWDOS/80 and TRSDOS.	7-8
7.14.	CMD"doscnd"	7-11
7.15.	CMD"F=POPS", CMD"POPR" and CMD"F=POPN"	7-12
7.16.	CMD"F=SASZ"	7-12
7.17.	CMD"F=ERASE" and CMD"F=KEEP"	7-12
7.18.	CMD"F",DELETE	7-13

7.19.	CMD"F=SWAP"	7-13
7.20.	CMD"F=SS"	7-14
7.21.	CMD"O"	7-14
7.22.	RENEW	7-17

Chapter 8. BASIC DISK I/O ENHANCEMENTS AND DIFFERENCES

8.1.	Introduction	8-1
8.2.	File Type	8-1
8.3.	File type differences	8-2
8.4.	Components of GET and PUT	8-3
8.5.	Fixed item file characteristics	8-7
8.6.	Marked item file characteristics	8-7
8.7.	OPEN	8-9
8.8.	GET	8-12
8.9.	PUT	8-14
8.10.	REMRA and REMBA	8-16
8.11.	Pseudo FIELD Function	8-17
8.12.	LOC Function	8-18
8.13.	I/O Error Recovery	8-19
8.14.	Additional notes about NEWDOS/80 DISK BASIC I/O	8-20

Chapter 9. ERROR CODES AND MESSAGES

9.1.	DOS Error Codes and Messages	9-1
9.2.	DISK BASIC Error Codes and Messages	9-2

Chapter 10. GLOSSARY10-1

Chapter 11. ERROR REPORTING, INCOMPATIBILITY HANDLING, AND PATCHING

11.1.	Introduction	11-1
11.2.	Incompatibility Handling	11-1
11.3.	Reporting of NEWDOS/80 Errors and Incompatibilities	11-2
11.4.	Format of NEWDOS/80 Zaps	11-2
11.5.	Zapping Procedure	11-4
11.6.	NEWDOS/80 Zap Distribution	11-5
11.7.	Initial Installation of Zaps	11-5
11.8.	Subsequent Installation of Zaps	11-6
11.9.	Diskette Update Service	11-6
11.10.	Zap Duplication.	11-7

Chapter 12. CONVERSION INFORMATION AND MISCELLANEOUS COMMENTS

12.1.	RBAs gain in respectability	12-1
12.2.	Converting from Ver. 1 to Ver. 2 on the Model I	12-2

12.3.	Converting from Ver. 1 to Ver. 2 on the Model III.....	12-5
12.4.	NEWDOS/80 Ver. 2 incompatibilities with TRSDOS Version 2.3.....	12-6
12.5.	NEWDOS/80 Ver. 2 incompatibilities with TRSDOS Version 1.3.....	12-7
12.6.	Miscellaneous Comments	12-8

Chapter 13. ZAPS (PATCHES)

APPENDIX A Discussion and example of NEWDOS/80 file routines

APPENDIX B Example of fixed and marked item file usage

APPENDIX C NEWDOS/80 Version 2.5 (Hard Disk System)

Index

1. INTRODUCTION

1.1. Registration.

As soon as you receive your NEWDOS/80, fill out and mail the registration card. Apparat will limit its assistance and patches (zaps) to registered owners only. In your communications with Apparat, always state your name, address and your NEWDOS/80's registration number. For Version 1 of NEWDOS/80 we had many complaints of not receiving zaps from users who had not sent in the registration card. Apparat does not require the owner to agree to anything when filling out the NEWDOS/80 Version 2 registration card; just let us know who you are.

1.2. Trademark Credits.

Throughout this manual, certain trademarked names will be used to refer to those trademarked products. Since our printers do not have the tm symbol, we will acknowledge the trademarked names here. If we have missed rendering an acknowledgement, please forgive us as we do not mean for any trademarked name to be used to refer to anything that the trademark holder does not mean it to refer to. In some cases, such as VTOS, the primary manual for that system shows the name trademarked but does not say who it is trademarked to.

1. TRS-80 is a registered trademark of Radio Shack, Inc.
2. TRSDOS is a registered trademark of Radio Shack, Inc.
3. VTOS is a registered trademark.
4. LDOS is a registered trademark of Lobo Drives International.
5. DOUBLER is a registered trademark of Percom Data Company, Inc.
6. SCRIPSIT is a registered trademark of Radio Shack, Inc.

1.3. What Is Apparat's DOS/80 Version 2?

Almost all disk based computer systems use a Disk Operating System (known as a DOS) to provide a software interface between the user program performing disk I/O and the actual disk drives and their controllers. Usually these operating systems perform many other functions as well such as controlling what user program is executing and the allocation of disk files and file space. Believe it or not, the primary function of a DOS is to make life easier for the computer users and programmers. NEWDOS/80 is one of a number of DOSs that operate with the TRS-80; in this case only the Model I and Model III are supported.

NEWDOS/80 Version 2 is the replacement for NEWDOS/80 Version 1 that was released in June of 1980 and for NEWDOS/21 that was released in March of 1979. NEWDOS/80 Version 2 is a disk operating system designed to operate on the TRS-80 Model I and the TRS-80 Model III. A particular NEWDOS/80 Version 2 master diskette is tailored to operate on only one of the two TRS-80 models; if you wish to operate on both the Model I and the Model III, you must purchase different NEWDOS/80's for each. The TRS-80 model being used must

have at least 32K of RAM and at least one 5 inch, single sided, 35 (40 for the Model III) track disk drive (mounted on drive 0). Model I NEWDOS/80 Version 2 is distributed on a 35 track, single sided, single density diskette, and Model III NEWDOS/80 Version 2 master diskette is distributed on a 40 track, single sided, double density diskette. You must have a disk drive capable of handling the master diskette.

NEWDOS/80 Version 2 for the Model I and NEWDOS/80 Version 2 for the Model III are mostly downward compatible with NEWDOS/80 Version 1, NEWDOS/21 and Model I TRSDOS 2.3, but it will be necessary to maintain certain programs with different copies for all four systems for incompatibilities do exist. NEWDOS/80 Version 2 is more incompatible with the Model III TRSDOS than it is with the Model I TRSDOS, and most programs and files will have to be maintained differently in the two systems. In the past, while TRSDOS was largely dormant, attempts were made to limit the incompatibilities between NEWDOS and TRSDOS, but now that TRSDOS is being actively updated more and more incompatibilities are appearing between the two systems. TRSDOS is going one way; NEWDOS/80 is going another. If this limits and eventually destroys NEWDOS's usefulness to the users, so be it.. NEWDOS cannot and should not exist to be a mirror image of TRSDOS; if the user wants that, then please use TRSDOS. NEWDOS was accidentally created in the huge vacuum left by Model I TRSDOS, has always incorporated features not in TRSDOS and, in Version 2, has not implemented many of the newer features of the Model III TRSDOS. Chapter 12, sections 12.1 through 12.5 give some of the incompatibilities of NEWDOS/80 Version 2 with NEWDOS/80 Version 1 and with the Model I and III TRSDOSs.

The DOS and DISK BASIC portions of NEWDOS/80 are total rewrites from that offered in NEWDOS/21. The requirement that the user purchase TRSDOS as a precondition of use of NEWDOS/21 is not required for NEWDOS/80. It is still recommended that the user purchase TRSDOS, and NEWDOS/80 users are expected to have purchased the TRSDOS manual and be knowledgeable of its contents as use of NEWDOS/80 assumes this user knowledge. Users of the EDTASM module are still required, as a precondition of use of NEWDOS/80's EDTASM, to have purchased Radio Shack's tape editor/assembler.

Though NEWDOS/80, Version 2 was tested more extensively than Version 1, there will still be errors, and many programs will require at least a zap to work with NEWDOS/80 Version 2. Error reporting procedures are discussed in chapter 11, and the outstanding zaps are in chapter 13.

1.4. Duplicate and Specify the System.

NEWDOS/80 is not a simple system. When the NEWDOS/80 user is ready to initially use NEWDOS/80, he/she should spend one to two hours studying the documentation before doing anything with the NEWDOS/80 diskette.

When ready, put a write protect tab on your NEWDOS/80 Version 2 master diskette. Then power up your computer, place the master diskette in drive 0 and press reset. The NEWDOS/80 banner should appear optionally followed by requests for date and time. If date and time are requested, please give realistic values. Next, NEWDOS/80 READY will be displayed to indicate DOS is waiting for something to do.

It is good practice to never mount on a disk drive the NEWDOS/80 master diskette except to make copies of the diskette and to very carefully apply mandatory zaps (see chapter 11). When zapping, you should first apply the zaps to a working Version 2 system diskette and test them out before applying them to the master diskette. Keep the master diskette stored away in a safe place; do not keep it in your NEWDOS/80 manual and do not use it in normal operations. Apparat will not replace a lost master diskette though it will, under the terms for the diskette update service offered in section 11.9, replace a damaged diskette.

Enter, via the keyboard, the DOS command:

```
LIB
```

A list of all the DOS library commands will be displayed to you. These commands are defined in chapter 2 with examples.

Enter the DOS command:

```
DIR,0,S,I
```

A list of all the files on the NEWDOS/80 Version 2 master diskette will be displayed. These files, except for NWD82V2/ILF and NWD82V2/XLF, are discussed in chapter 5.

Enter the DOS command:

```
SYSTEM,0
```

NEWDOS/80 offers the user certain system options, which are specified via the DOS library command SYSTEM (see section 2.46) and activated during each computer reset. The DOS command SYSTEM,0 you just executed has displayed the state of all SYSTEM options, and you should compare these value carefully against the specifications. You may decide that your system is to use different SYSTEM specifications. You may change them now if absolutely necessary; otherwise you should wait until after you have made a few backup copies of the master diskette. Whenever you decide to update the master diskette, don't forget to take off the write protect tab.

Enter the DOS command

```
PDRIVE,0
```

NEWDOS/80 can operate with a limited mixture of disk drive and interface types. The characteristics of each of the physical drives 0 - 3 must be specified to the system via the DOS library command PDRIVE (see section 2.39). These characteristics are then read by DOS during each computer reset. The PDRIVE command you just executed has displayed the existing drive specifications plus 6 pseudo drive specifications. You may want to change the specifications for one or more drives. You may do so now if absolutely necessary; otherwise you should wait until you have a few backup copies of the master diskette.

Now you must make three or more copies of the NEWDOS/80 Version 2 master diskette. If possible, perform these initial backups without changing any of the SYSTEM or PDRIVE parameters. If not possible, change them the minimum

necessary and do a reset when done. You should carefully study sections 2.14, 2.39 and 2.46.

NEWDOS/80 does NOT have a BACKUP module; format 5 or 6 of DOS library command COPY (see section 2.14) is used instead. For each of the backups you are about to do, the master diskette is both the system and the source diskette while the destination diskette is the diskette to contain the new working copy of the NEWDOS/80 system. Some examples of the COPY command you will use to make copies of the NEWDOS/80 Version 2 master diskette are:

COPY,0,0,,FMT,USD For a single drive system where the master and copy diskettes have the same PDRIVE characteristics.

COPY,0,1,,FMT,USD For a multiple drive system where the master and copy (mounted on drive 1) diskettes have the same PDRIVE characteristics.

COPY,0,0,,FMT,USD,CBF,DPDN=4 For single drive system wherein the destination diskette has PDRIVE characteristics different from the master diskette. You must have previously altered the master diskette PDRIVE specification for drive 4 (remember to use the A option or to reset the computer after changing the drive 4 specification).

COPY,0,1,,FMT,USD,CBF For a multiple drive system where the drive 1 drive will be moved to drive 0 after the copy and the destination drive has different PDRIVE characteristics than does the current drive 0. You must have previously altered the master diskette's PDRIVE specification for drive 1.

Each system diskette has its own set of SYSTEM and PDRIVE characteristics. Therefore, for each working copy of NEWDOS/80 Version 2 you make, after the copy is completed, you need to set that system diskette's SYSTEM and PDRIVE characteristics for the operating conditions it is to operate with.

The NEWDOS/80 owner is authorized to make as many copies as necessary of the NEWDOS/80 diskette or individual programs thereon for his/her own personal use. NEWDOS/80 owners and users are specifically prohibited from copying the NEWDOS/80 diskette or individual programs thereon for use by others. See COPY, formats 2 and 4, in section 2.14.

1.5. Apply Outstanding Zaps.

Before your NEWDOS/80 is ready to run user programs, review the outstanding zaps to both NEWDOS/80 modules and to other modules (such as EDIT/CMD and SCRIPSIT) that require patches to work properly with NEWDOS/80. Chapter 11 explains how to apply zaps (patches), and with your NEWDOS/80 should have come a chapter 13, which contains the zaps. If part or all of chapter 13 is not in the proper place in the manual please put it there. Mandatory zaps must be applied; optional zaps are at user discretion.

Mandatory zaps to NEWDOS/80 modules should be applied to all copies of the NEWDOS/80 Version 2 master diskette and to the NEWDOS/80 Version 2 master diskette. DO NOT start applying the zaps until you have at least 2 or 3 good backup copies made of the NEWDOS/80 diskette.

1.6. Commence Using NEWDOS/80.

Once all backup copies of the NEWDOS/80 Version 2 system are made, zaps applied, system options and drive characteristics specified, you are now ready to use NEWDOS/80.

Put away the master diskette and mount in drive 0 one of the system diskette just made. -Then press reset to re-initialize DOS using the new diskette. NEWDOS/80 READY will then appear. The user may now type in a DOS command, which is either a-DOS library command as discussed in chapter 2 or the name or name/ext of a user program to be loaded and run. If a user program does not have a name extension, name extension CMD is assumed. Examples:

BASIC causes the load and execution of program BASIC/CMD.

SCRIPSIT/LC causes the load and execution of program SCRIPSIT/LC.

If the DOS library command or the user program requires or allows for parameters within the DOS command, one or more spaces or a comma must follow the command name and precede the parameter(s). Examples:

BASIC,5,65000
DIR 1 A

For virtually all programs to be executed under NEWDOS/80, there are instructions on how to use the program that usually comes with the program when you buy it. For NEWDOS/80 program modules, the instructions are in chapter 6 except for BASIC, which is covered in chapters 7 and 8.

Those users upgrading from NEWDOS/80 Version 1, NEWDOS/21 or TRSDOS to NEWDOS/80 Version 2 should read sections 12.1 through 12.5 carefully.

1.7. Apparat Thanks Its Beta Testers.

Over forty persons throughout the United States and Canada were involved in the testing of NEWDOS/80 Version 2, finding errors, suggesting enhancements and providing criticism. Apparat and the NEWDOS/80 author thank each one of these beta testers for the long hours spent working with the three beta releases that were sent out. It is Apparat's policy that each beta tester receives a complimentary copy of the final release of NEWDOS/80 Version 2.

2. DOS LIBRARY COMMANDS

2.1. Notation Conventions and General Information.

All DOS commands terminate with an ENTER. In subsequent specifications, the ENTER is not shown, but the user is to supply it.

DOS commands are limited to a total of 80 characters, including the concluding ENTER.

[] A set of brackets are used to enclose an optional parameter. When using the optional parameter in a DOS command, the [] are not included.

Example:

```
[,PROT=xxx][,ASE=yn][,ASC=yn]
could be coded as
    ,PROT=READ,ASC=N
```

Uppercase A - Z and non-alphanumeric characters are to be included exactly as shown. See the above example.

Lower case letters or words with or without trailing decimal digits. These represent prototype values for which the user is to substitute the appropriate actual values. See the above example.

In some cases where the prototype will be replaced by one and only one character, the prototype word contains, in lower case, all the characters legal for that value. This helps serve as a reminder of which characters are legal replacement for that prototype value. For example, if ASC=Y and ASC=N are the only two legal ASC values, then the prototype will usually be written as ASC=yn.

Where commas are used in DOS commands, they may be replaced by one or more consecutive spaces.

Numeric values without a suffixed H are considered decimal values unless otherwise specified. Hexadecimal values must be suffixed with an H unless otherwise specified. Example:

4000H and 16384 are the same value.

When specifying a disk file, the term 'filespec' is used. A filespec is of the form:

```
name1[/ext1][.password1][:dn1]
```

Parameters must be specified in the above order.

name1 is the file's name consisting of 1 - 8 chars of which the first must be A - Z and the others A - Z or 0 -9.

ext1 is the name extension (i.e., CMD, BAS, OBJ, CIM, TXT, DOC, COM, etc.) which classifies a file. A file need not have a name extension, but if it does it must be 1 - 3 chars of which the first must be A - Z and

the others A - Z or 0 - 9. If a file has a name extension, all filespecs referencing the file must include the name extension, unless a default name extension is provided for (i.e., /CMD).

password1 is 1 - 8 chars of which the first must be A - Z and the others A - Z or 0 - 9. Password1 is the value given to both the access and update passwords for a file when it is created. Password1 is value used in password checking when an existing file is opened. Password1 is required in a filespec if passwords are enabled and the file has passwords assigned; otherwise, it is not.

dn1 is the drive # of the drive which has the diskette containing the file. Examples:

```
MYFILE80/BAS.YOURPW80:0
MYFILE:3
YOURFILE.YOURPW
```

NEWDOS/80 will accept lowercase in all DOS library commands and any further input that might be queried for.

For each DOS library command, the command keyword is stated along with a brief definition. Next, if the command is allowed parameters, a prototype of the command is given, listing all required and optional parameters. Next comes explanations of the command, parameters and options. Lastly, some examples of the DOS command are given.

For documentation ease, the prototype command is sometimes shown spread over multiple lines in this document; however, the user should consider each command as one contiguous statement.

Unless otherwise stated, a DOS library command is executable under MINI-DOS (see section 4.2).

NEWDOS/80 differs from TRSDOS in NOT using parenthesis to enclose parameters. In NEWDOS/80 version 1, parenthesis around the operands were optional for BREAK, CLOCK, DEBUG, DIR, PROT, and VERIFY; they are NOT allowed in version 2.

In the same vein, version 1 allowed the keywords ON or OFF to be used instead of Y or N in the DOS commands BREAK, CLOCK, DEBUG and VERIFY; this is NOT allowed in version 2.

2.2. APPEND Append one file onto the end of another.

```
APPEND,filespec1,[TO,]filespec2
```

This command will append the file filespec1 onto the end of the file filespec2. The EOF from file filespec2's directory FPDE determines the point at which file filespec1 is appended. This may be trouble if file filespec2 had explicit EOF characters, such as in BASIC program files or assembler source files.

File filespec1 is not altered. The original contents of file filespec2 are not altered; the file is only added to.

APPEND is not executable under MINI-DOS.

APPEND examples:

1. APPEND,XXX:1,YYY/DAT:0 The contents of file XXX on drive 1 are appended onto the end of file YYY/DAT which is on drive A
2. APPEND AAA TO BBB The contents of file AAA are appended onto the end of file BBB. DOS searches the currently mounted diskettes to find both files.

2.3. ATTRIB Assign attributes to a file.

```
ATTRIB,filespec1[,INV][,VIS][,PROT=xxx][,ACC=password1][,UPD=password2]
[,ASE=e][,ASC=c][,UDF=u]
```

This command assigns attributes to the filespec1 file. At least one of the optional parameters must be specified.

If passwords are enabled in your system, then filespec1 must specify the existing update password, if any, for that file.

INV gives the file the invisible attribute. Unless the I option is specified in DIR, the file will not be listed by AIR.

VIS takes away the invisible attribute, whether the file had it or not.

PROT=xxx specifies the access level to be used during file I/O if passwords are enabled (see system option AA) and the access, not the update, password was used to open the file. The levels are defined for values of xxx as:

LOCK Level 7. No access allowed to the file at all, except by the system's overlay loader.

EXEC Level 6. Access allowed only to execute the file as a program. BASIC will require either RUN or LOAD with R option, and will disable the BREAK key, thereby preventing the user from stopping the RUN and disallowing direct statement execution.

READ Level 5. Access allowed for execute or to read the file's contents.

WRITE Level 4. Access allowed for execute, read or write of the file.

RENAME or NAME Level 2. Access allowed for execute, read, write or to rename the file.

KILL Level 1. Access allowed for execute, read, write, rename or to kill the file.

FULL Level 0. All operations are allowed on the file.

ACC=password1 Password1 is assigned as the access password for the file. If null, a value of all blanks is assumed; otherwise the value must be 1 - 8 characters with the 1st = A - Z and the others A - Z or 0 - 9. Assigning the access password via this parameter of ATTRIB is the only way that will enable use of the PROT=xxx protection and then only if the access password is different from the update password. If a password is specified when the file is created, it is assumed both the update and the access password, and the update password has priority at open time. If passwords are enabled, the password specified in the filespec at open time is not the update password, and it is the access password, the current protection level is stored into the FCB for later use by the DOS read, write, load, etc. routines. Subsequently, if an access is attempted in violation of the access level, 'ILLEGAL ACCESS TRIED TO A PROTECTED FILE' error will occur.

UPD=password2 Password2 is assigned as the update password for the file. The update password is of the same configuration as the access password. During file open where passwords are enabled, the password specified in the filespec is checked first against the file's update password. If they match, then FULL access is allowed to the file.

ASE=e where e is either Y or N. This parameter has been added to allow DOS to automatically allocate diskette space to a file if ASE=Y or to disallow further allocation if ASE=N. ASE=Y is the default condition when a file is created.

ASC=c where c is either Y or N. This parameter has been added to allow DOS to automatically deallocate file diskette space beyond the EOF during a CLOSE operation if ASC=Y is specified, and to disallow this deallocation if ASC=N. ASC=Y is the default setting when a file is created.

UDF=u where u is either Y or N. This parameter has been added to mark the file as updated if UDF=Y is specified or to clear the updated mark if UDF=N is specified. The DOS system marks a file as updated whenever it is about to update a sector to that file and it finds the file's directory entry not marked as updated.

ATTRIB command examples:

1. ATTRIB,XXX/CMD:1,UPD=ZXCVB,ACC=NMLKJ,PROT=EXEC Assigns to file XXX/Crib located on drive 1 the update password ZXCVB, the access password NMLKJ and protection level 6 which allows the program to be executed but not read or written to. Since the filespec XXX/CMD:1 did not specify a password, we must assume that either password checking was disabled (SYSTEM option AA=N) or the file did not have an update password prior to the ATTRIB command.

2. ATTRIB YYY/DAT.QZBV INV ASE=N ASC=N UDF=N This command tests if file YYY/DAT has update password QZBV, and if so, assigns the file the invisible attribute, flags that extra space allocation and excess space deallocation are not to be allowed, and lastly clears the file's updated flag.

2.4. AUTO Define the DOS command to be executed at reset.

AUTO[,doscmd]

This command allows the user to specify a 1 - 31 character DOS command to be invoked automatically at reset time. This command is stored in the last 32 bytes of GAT sector of the current system diskette.

If doscmd is not specified, then a null command is stored in the GAT sector to indicate to reset/power-on that no AUTO command exists.

If SYSTEM option AB = N and BC = Y, by pressing ENTER during reset, the auto command in the GAT sector will be ignored, and the system will go to DOS READY.

AUTO is useful to the user who usually executes the same command or chain of commands (see CHAIN, sections 2.9 and 4.3, and DO, section 2.19) at reset time. By setting system option AB=Y or BC=N, the user is forced to this command or chain of commands, thus allowing the persons) controlling a computer to restrict the activity of users of the computer.

AUTO command examples:

1. AUTO BASIC RUN"XXX/BAS" causes subsequent reset/power-ons to activate BASIC and to start the execution of the BASIC program XXX/BAS.
2. AUTO DO RS ACTIONcauses subsequent reset/power-ons to activate chaining from file RS ACTION/JCL, thus executing the DOS and other program commands contained therein.
3. AUTO causes subsequent reset/power-ons to go to the normal DOS READY, awaiting the next DOS command to be inputted from the keyboard.

2.5. BASIC2 Activate non-disk BASIC (Model I only).

This command puts the system into non-disk BASIC. NEWDOS/80 is no longer in the system.

2.6. BLINK Enable/disable cursor blinking.

BLINK[,yn]

BLINK or BLINK,Y Blinking of the display cursor is turned on.

BLINK,N Blinking of the display cursor is turned off.

SYSTEM option BH can be used to set the cursor blinking state at reset/power-on.

2.7. BOOT Reset the computer.

On the Model I, this command deselects the drives and then executes Z-80 instruction HALT, which causes both a hardware and a software reset. For the Model III, since HALT does not cause a hardware reset, this instruction causes a jump to location 0 to execute a software reset.

2.8. BREAK Enable/disable the BREAK key.

`BREAK(,yn]`

`BREAK` or `BREAK,Y` The BREAK key is enabled as a normal input key (hexadecimal code 01) until the next normal DOS READY, when it is set according to system option AG.

`BREAK,N` The BREAK key is disabled as a normal input key until the next normal DOS READY, when it is set according to system option AG.

The BREAK command is useful for those programs that want the BREAK key enabled, and enables it via a DOS-CALL (vector 4419H). The same applies to programs that definitely want BREAK disabled. NOTE: Executing BREAK from DOS READY is useless since the immediate return to DOS READY resets the BREAK key according to system option AG.

In NEWDOS/80 the BREAK key may also be enabled by storing a 0C9H byte in Model I location 4312H (Model III location 4478H) and may be disabled by storing a 0C3H byte in that location. In NEWDOS/80 version 1, the break key was also manipulated by changing bit 4 of location 4369H (Model I only); in version 2 for the Model I, setting or clearing this bit does nothing and is harmless. However, programs on the Model III must NOT alter that bit, as that location is now in the system buffer.

2.9. CHAIN Shift to keyboard input from disk.

`CHAIN,filespec1[,sectionid]`

DOS command DO performs exactly the same as CHAIN.

The purpose of the CHAIN command is to cause a predefined set of characters to be treated as input from the keyboard. This predefined set of characters has been previously stored in the file `filespec1`.

The CHAIN command places NEWDOS/80 in chaining mode, if not already there. The file `filespec1` is opened. If `sectionid` is not specified, the file is positioned at the beginning. If `sectionid` is specified, the file is searched for the matching `sectionid` record, leaving the file positioned at the byte following the section ID record.

Subsequently, input that is supposed to come from the keyboard comes from the chain file until chaining is terminated by the encounter of either end of

file or end of section or until chaining is temporarily halted by the execution of the DOS command CHNON,N.

Keyboard data is input from the chaining file in one of two modes.

If SYSTEM option AT = N, chaining operates in record mode. In this mode, whenever NEWDOS/80, BASIC or any program requests a new record from the keyboard via the standard ROM keyboard record input routine at 05D9H, the record will come from the chain file. Any other requests for keyboard input are honored from the keyboard and not the chain file.

If SYSTEM option AT = Y, chaining operates in byte mode. In this mode, all requests for keyboard input characters via the standard keyboard input routine are honored from the chain file.

The CHAIN command may be issued via DOS-CALL or via BASIC's CMD function. When so, DOS does not immediately return to the calling program but instead continues to execute commands from the chain file until either end of file, end of section, command CHNON,N or command CHNON,Y is encountered.

CHAIN is not legal under MINI-DOS.

The chain file creator/maintainer is responsible for assuring that chaining does not create impossible situations for the system or user programs.

NEWDOS/80 cannot have more than one chain file active at a time. If the new DOS command from the current chain file is itself a CHAIN or DO command, processing in the current file ceases and the new chain file is opened, becoming the new current chain file.

When the system opens a chain file, name extension in the filespec defaults to JCL if the filespec doesn't give one.

CHAINING is discussed further in section 4.3.

CHAIN or DO command examples:

1. CHAIN,XXX:0 Chaining starts at the beginning of file XXX/JCL:0.
2. DO YYY/CHN:1,QQQ Chaining starts at the first byte of the chain section named QQQ within file YYY/CHN.

2.10. CHNON Alter chaining state.

CHNON,ynd

The CHNON command is used during chaining. An error will occur if a chain file is not currently open. A CHNON command should not be the last entry in an unsectioned chain file or the last entry in a chain file section as the command will be meaningless.

CHNON,N The current position within the chain file is remembered and chaining is temporarily suspended so that subsequent keyboard characters to come from

the keyboard. If chaining was being done under DOS-CALL, the current DOSCALL level is exited.

CHNON,Y causes subsequent keyboard characters to come from the chain file, starting at the current position within the chain file. If CHNON,Y was executed as a DOS-CALL, the current DOS-CALL level is exited.

CHNON,D causes subsequent keyboard characters to come from the chain file, starting at the current position within the chain file. If CHNON,D was executed as a DOS-CALL, DOS remains at that level and executes subsequent commands from the chain file until either CHNON,Y or CHNON,N or end of section or end of file is encountered.

See sections 2.9 and 4.3 for further discussion of chaining.

2.11. CLEAR Clear user memory routes, timer and logical enqueues.

```
CLEAR[,START=addr1][,END=addr2][,MEM=addr3]
```

The CLEAR command performs the following functions:

1. Performs ROUTE,CLEAR DOS command function.
2. Dequeues all user routines in the timer interrupt routine chain that were enqueued by the 4410H (Model I) or 447BH (Model III) call to DOS. This includes turning the clock display off.
3. Dequeues all *name routines that were enqueued by a 4461H call to DOS. This includes the NEWDOS/80 spooler, if active, but not its graceful termination. The spooler, if in use, should be fully terminated before executing CLEAR.
4. Resets HIMEM to addr3 or, if addr3 not specified, to the highest memory address.
5. Zeroes memory from addr1 or 5200H, which ever is greater, through addr3 or HIMEM, whichever is lower. addr1 must be greater than or equal to 5200H and less than or equal to addr3.

CLEAR command examples:

1. CLEAR,START=6000H,MEM=0DFFFFH All routes are cleared, and all timer and *name routines dequeued. HIMEM is set to 0DFFFFH. The main memory between 6000H and 0DFFFFH is zeroed.
2. CLEAR All routes are cleared, and all timer and *name routines dequeued. HIMEM is set to the highest main memory location, and all memory from 5200H to HIMEM is zeroed.

2.12. CLOCK Display the time every second.

CLOCK[,yn]

CLOCK or CLOCK,Y The current value of the clock is displayed every second in positions 53-60 of the display's top line in HH:MM:SS format.

CLOCK,N The displaying of the clock ceases.

Users are warned that the clock will continuously lose time. There is no hardware clock in the sense of seconds, minutes and hours. Computation of clock time is done from the 25ms interrupt chain in the Model I (in the Model III, it is done in the ROM from the timer interrupt). Whenever the interrupts are left off for more than 25ms (33 or 40 ms on the Model III), one or more interrupts are lost and for each one lost, the clock loses 25ms (33 or 40 ms on the Model III). Lost interrupts are very frequent when disk I/O is being done, is massive when tape I/O is done, and can also be frequent if other routines hung off the 25ms chain are more than a few milliseconds long.

2.13. CLS Clear the display.

CLS simply clears the display, resetting it to 64 character mode. On the Model III, reserved top display lines are not cleared.

2.14. COPY

The COPY command is used to copy a single file, multiple files or a full diskette. COPY has 6 formats:

1. COPY,filespec1[,TO],filespec2[,SPDN=dn3][,DPDN=dn4]
2. COPY,\$filespec1[,TO],filespec2[,SPDN=dn3][,DPDN=dn4]
3. COPY,[:]dn1,filespec1[,TO],filespec2[,SPDN=dn3][,DPDN=dn4]
4. COPY,[:]dn1,\$filespec1[,TO],filespec2[,SPDN=dn3][,DPDN=dn4]
5. COPY,[:]dn1[=tc1][,TO],[:]dn2[=tc2],mm/dd/yy[,Y][,N]
[,NDMW][,FMT][,NFMT][,SPDN=dn3][,DPDN=dn4][,SPW=password1]
[,NDPW=password3][,DDND][,ODN=name1][,KDN][,KDD][,NDN=name2]
[,SN=name3][,USD][,BDU][,UBB]
6. COPY,[:]dn1[,TO],[:]dn2[=tc2],mm/dd/yy,CBF[,Y][,N]
[,USR][,/ext][,UPD][,ILF=filespec3][,XLF=filespec4][,CFWO]
[,NDMW][,FMT][,NFMT][,SPDN=dn3][,DPDN=dn4][,SPW=password1]
[,ODPW=password2][,NDPW=password3][,DDND][,ODN=name1]
[,KDN][,KDD][,NDN=name2][,SN=name3][,USD][,UBB]
[,DDSL=ln1][,DDGA=gc1]

The COPY command has been significantly changed in NEWDOS/80 version 2; all users, new and old, should carefully read this section.

COPY cannot be executed under MINI-DOS; however for simple single file copies, DOS library command MDCOPY is available.

dn1 and dn2 are drive numbers and may be equal. The colon preceding dn1 and/dn2 is optional.

Filespec1 is the source file's filespec. Filespec2 is the destination file's filespec.

Filespec1 prefixed with \$ means that either the source or the destination file or both are to be on drive 0 and are on diskettes) that either (1) do not contain a NEWDOS/80 system identical to the one on drive 0 when COPY was initiated, (2) do not contain a NEWDOS/80 system, or (3) contain no system at all.

During processing for formats 2, 3, 4, 5 and 6, the system may ask for various diskette mounts; do what the prompts ask!!

1. When prompted for the system diskette, mount the NEWDOS/80 diskette that was on drive 0 at the start of the COPY command execution.
2. When prompted for the source diskette, mount the diskette containing file filespec1 (formats 1, 2, 3 and 4) or the data to be copied (formats 5 and 6).
3. When prompted for the destination diskette, mount the diskette to contain file filespec2 (formats 1, 2, 3 and 4) or to receive the data being copied (formats 5 and 6).

SPDN=dn3 Source PDrive Number. SPDN=dn3 tells the system that for all source drive I/O, the system diskette's PDRIVE specifications (see DOS command PDRIVE, section 2.37) for drive dn3 are to be used instead of the source drive's normal PDRIVE specifications. dn3 is a value 0 to 9, referring to a drive number listed by the PDRIVE command.

DPDN=dn4 Destination PDrive Number. DPDN=dn4 tells the system that for all destination drive I/Os, the system diskette's PDRIVE specifications for drive dn4 are to be used instead of the destination drive's normal PDRIVE specifications. dn4 is a value 0 to 9 referring to a drive number listed by the PDRIVE command.

Note that use of SPDN and DPDN for a drive 0 single drive COPY (formats 4, 5 or 6) means that three different PDRIVE specifications (one for the system diskette, one for the source diskette and one for the destination diskette) will apply during the COPY even though only one drive is used.

Format 1 is the single file copy. It is used to copy the contents of file filespec1 to file filespec2. The diskettes involved in the COPY must already be mounted; the system gives no mount prompts. The contents of file filespec1 are not altered. The previous contents of file filespec2, if any, are lost. If the leading part of filespec2 equals that of filespec1, filespec2 may be shortened by leaving off the leading part, the remainder of filespec2 starting with / or . or :. For example:

```
COPY,USERFILE/DAT:0,TO,USERFILE/DAT:1
```

can be shortened to:

```
COPY,USERFILE/DAT:0,TO,:1
```

Remember, the keyword TO is optional, and spaces may be used instead of commas. Thus, the command could be written:

```
COPY USERFILE/DAT:0 :1
```

Format 2 is the same as format 1 except that the \$ sign prefixed onto filespec1 indicates that a conflict exists with drive 0, the system drive, and DOS will prompt for the proper diskettes to be mounted on drive 0. If the source and destination drive numbers are both zero but the source and destination files are on separate diskettes, use format 4 instead of format 2.

Format 3 again is similar to format 1, except that the user has only 1 drive available for the copy and file filespec1 resides on a diskette different from that of file filespec2. Neither filespec can specify a drive number. DOS will prompt for the mount of the source and destination diskettes as they are needed. If drive 0 is specified, both the source and destination diskettes must contain a NEWDOS/80 system identical to the one mounted on drive 0 at the start of the COPY command; otherwise use format 4.

Format 4 performs similar to format 3 except that either file or both reside on diskettes with different NEWDOS/80 systems, non-NEWDOS/80 systems or no systems at all. DOS will prompt for the mount of the system, source and destination diskettes as they are needed. Format 4 should only be used when dnl equals otherwise you are wasting time with diskette swaps that are not needed.

Formats 2 and 4 allows suppliers of programs, whether free or purchased, to send their program products on diskettes that do not contain NEWDOS systems. Aside from the supplier's programs and/or data files, the diskette need only contain the directory and the BOOT/SYS file, both created on each diskette during formatting. Suppliers must not include a NEWDOS system on their diskettes unless they have made explicit arrangements with Apparatus.

NEWDOS/80 does not have a diskette BACKUP program. Instead, either formats 5 or 6 is used. Format 5 is a full diskette sector by sector copy without concern for the number and type of files. Format 6 copies some or all of the source diskette's files onto the destination diskette. Of the two, for the same amount of data transmitted, format 5 is faster while format 6 allows greater variation between source and destination diskette types and tries to reassign files to contiguous space.

Format 5 is a full diskette copy. The default specifications for the two drives are the PDRIVE specifications currently being used by DOS. The drives must have the same number of sectors per track, granules per lump and sectors per granule (five is the current NEWDOS/80 standard); otherwise format 6 must be used. The destination diskette may have more tracks than the source; if so, the destination directory is adjusted to account for the extra free granules (not done if BDU option specified). Format 5 options are defined as follows:

=tc1 DOS is to use the value tc1 as the source diskette's track count during the COPY rather than the source drive's default value.

=tc2 DOS is to use the value tc2 as the destination diskette's track count during the COPY rather than the destination drive's default value.

mm/dd/yy is the date to be placed in the destination diskette date field. The mm/dd/yy may be nulls and if so the system date is used. The only time mm/dd/yy may be entirely left out of the format 5 COPY command is when the command has only the two drive number parameters (example: COPY 0 1). Otherwise mm/dd/yy must be the 3rd parameter even if it is null or to be overridden by either the KDD or the USD parameter. If the mm/dd/yy is null, this must be so indicated by separating commas (not spaces)

(example: COPY 0 1,,FMT CBF).

Y The user doesn't care what was previously on the destination diskette. Y is mutually exclusive with N, ODN, ODPW, DDND, KDN or KDD. Y is the default (for COPY) if none of its mutual exclusions are specified.

N At the start of the COPY or FORMAT the destination diskette must not contain recognizable data, i.e., should be in a bulk erase state. COPY will be terminated if the diskette is found to contain data. N is mutually exclusive with Y, ODN, ODPW, DDND, KDN or KDD.

NDMW No Diskette Mount Waits. DOS is to assume that all needed diskettes are already mounted on the specified drives. No mount prompts or error prompts are displayed. If an error occurs that otherwise would have caused a prompt, the copy will be terminated. If NDMW is specified and neither FMT nor NFMT are specified, FMT is assumed. NDMW is intended for use when COPY (or FORMAT) is invoked via DOS-CALL (i.e., from BASIC) and the calling program does not want operator interaction. Since NDMW causes the COPY or FORMAT to bypass error and disk mount queries, it is recommended that NDMW normally not be used when the operator is keying in the COPY (or FORMAT) command.

FMT Format. DOS formats the destination diskette before copying data. FMT is mutually exclusive with NFMT. If neither FMT or NFMT is specified and NDMW was not specified, the operator will be queried 'FORMAT DISKETTE? (Y OR N)'. If neither FMT or NFMT is specified and NDMW was specified, FMT is assumed.

NFMT No Format. DOS does not format the destination diskette before copying data. The user must assure that the destination diskette is already formatted correctly. NFMT is mutually exclusive with FMT.

SPW=password1 Source Password. If passwords are enabled (system option AA = Y) and system option AR = N, then COPY requires a source diskette master password match. If password1 does not match the source diskette's password, the copy function will be terminated.

PDPW=password3 New Destination Password. Password3 must conform to rules for passwords and is assigned as the destination diskette's new password. NDPW is mutually exclusive with BDU.

DDND Display Destination old Name and Date. The destination diskette's old name and date are prompted to the display, allowing the operator to decide whether or not to proceed with the copy. DDND is mutually exclusive with Y, N, and NDMW.

ODN=namel Old Destination Name. If the destination diskette's old name is not equal to namel, then the system prompts, allowing the

operator to decide whether to proceed with the copy. ODN is mutually exclusive with Y, N and NDMW.

KDN Keep Destination diskette Name. The destination diskette keeps its old name rather than receive the source diskette's name. KDN is mutually exclusive with Y, N, BDU and NDN.

RDD Keep Destination diskette Date. The destination diskette keeps its old date rather than receive the mm/dd/yy parameter from the COPY command. KDD is mutually exclusive with Y, N, BDU and USD.

NDN=name2 New Destination Name. The destination diskette takes name2 as its name, rather than receive the source diskette's name. Name2 must conform to the specification for diskette names. NDN is mutually exclusive with BDU and KDN.

USD Use Source Date. The destination diskette uses as its date the source diskette's date, rather than receive the mm/dd/yy parameter from the COPY command. USD is mutually exclusive with KDD and BDU.

SD=name3 Source diskette Name. If the source diskette's name is not equal to name3, a prompt is issued, allowing the operator to decide whether or not to proceed with the copy.

BDU Bypass destination Directory Update. Aside from simply copying the source sectors onto the destination diskette, the format 5 COPY also updates the boot and PDRIVE data in the destination file BOOT/SYS and, as necessary, the name, date, password and extra granule information into file DIR/SYS. There are times, however, when this file updating is not wanted, and by specifying option BDU these updates are bypassed. BDU is useful when the source diskette has a bad directory, has a non-standard directory (such as a TRSDOS Model III directory) or has no directory at all or when the user wants a full diskette copy with no alterations. BDU is mutually exclusive with KDN, NDN, NDPW and USD.

UBB Use Big Buffer in NEWDOS/21 and NEWDOS/80 version 1, COPY was restricted to using main memory below 7000H unless it was a two diskette, single drive COPY, in which case all of memory to HIMEM was used. If a user wanted to force the usage of all memory to HIMEM, the UBB parameter had to be specified. However, in NEWDOS/80 version 2, all of main memory to HIMEM is used unless the COPY was invoked under DOS-CALL (i.e., from BASIC), in which case only main memory below 7000H is used. Thus, in NEWDOS/80 version 2, UBB is a useless parameter left in existence only for upward compatibility from Version 1.

Format 6 is the multiple file COPY and is distinguished from format 5 by the inclusion of the CBF (Copy By File) option. Though format 5 is the faster way to backup a diskette, format 6 offers more flexibility, allowing files to be copied between diskettes and drives of widely varying characteristics. The choice of files to be copied can be limited by the combined effect of options USR, /ext, UPD, ILF, XLF and CFWO; if one or more criteria are specified, only those files satisfying all the criteria are copied. Format 5's options, except BDU, are used in format 6 as well as the following additional options.

If NFMT is specified, then none of Y, N, KDN, KDD, NDN, BDU, USD, NDPW, DDSL, DDGA or tc2 may be specified, ODPW may be required, and system

files are not copied unless already existent in the destination file directory.

If NFMT is not specified, then the destination file is formatted as if the command was FORMAT, including establishing BOOT/SYS and AIR/SYS. Then, before any files are copied, all files to be copied are entered into the destination diskette's directory. This is necessary as system files must occupy the same directory FPDEs in order for DOS to work at all.

CBF Copy By File CBF, required for and used only in format 6, indicates the copy will be done by files rather than in straight sequential order of diskette sectors.

USR copy user files. Only user files are copied; system and invisible files are excluded.

/ext copy files having name extension ext. Only files with name extension ext are copied. ext is a 0 to 3 character name extension. Examples of this parameter are /CMD, /, /BAS, /X.

DPD copy updated files. Only files that have the updated flag on in the source diskette directory are copied. This flag is turned on by the standard DOS sector write routine to indicate that at least one sector has been written or re-written to this file since the last time the updated flag was cleared. This flag is turned off by specific request via the PROT or ATTRIB commands and is NOT turned off by COPY. Since the standard DOS sector write routine is used to write the file's sectors to the destination diskette, the updated flag is turned on for the copied destination files.

ILF=filespec3 Include List File Filespec3 specifies a file containing a list of files to be copied. If a file is not in the list, it is not copied. It is not an error if an included file is not on the source diskette. Within the list, each file to copied is specified by its name/ext followed by a EOL char (0DH). If a specification begins with a semi-colon, it is bypassed as a comment. Each specification, except comment, is limited to a maximum of 13 characters, including the EOL. On reading, the file's bytes are modulated into the ASCII range 0 to 127. The file can be made using SCRIPSIT, but the user must assure that no characters other than null (00H) follow the last EOL character; SCRIPSIT tends to leave extraneous characters so a delete-to-end-of-text should be done. ILF is mutually exclusive with XLF.

XLF=filespec4 Exclusion List File. The file filespec4 is the same structure as specified for ILF above and specifies the files to be excluded from the COPY. It is not an error if an excluded file is not on the source diskette. XLF is mutually exclusive with ILF.

CFWO Check File With Operator. For the qualifying files, DOS asks the operator, one file at a time, if the file is to be copied to the destination diskette. Reply Y if the file is to be copied, reply N if not to be copied, reply R if to restart entire CFWO query sequence, or reply Q if no more files to be copied. No files are copied until the querying is completed.

ODPW=password2 Old Destination diskette Password. If NFMT is specified, if passwords are enabled and if system option AR = N, then copy requires a destination diskette password match. If password2 does not match the destination diskette's password, the copy is terminated.

DDSL=ln1 Destination diskette Directory Starting Lump. Formatting will start the directory on the 1st sector of lump IS if DDSL is specified; otherwise the default starting lump number for the drive (see PDRIVE command) will be used: DDSL is mutually exclusive with NFMT.

DDGA=gcl Destination diskette Directory Granule Allocation. Formatting will allocate gcl (value 2 - 6) granules to the directory if DDGA is specified; otherwise it will assign the default # of granules for that drive (see PDRIVE command). DDGA is mutually exclusive with NFMT.

If during a format 6 COPY, the destination diskette has insufficient space to contain a file, "DISKETTE FULL = name/ext" is displayed and the destination file's EOF is set to 0. Though EOF is set to 0, any space the file may have allocated to it is not deallocated.

A single drive format 5 or 6 COPY cannot be executed under DOS-CALL (i.e., from BASIC) since COPY under DOS-CALL restricts itself to main memory below 7000H and this would necessitate too many diskette swaps.

During a COPY or FORMAT where NDMW was not specified, pressing right arrow at any time will cause the function to pause, awaiting ENTER to continue or up-arrow to cancel. Pressing up-arrow at any time will terminate the function; however, be careful as the state of the destination diskette will be unknown, especially if the cancel comes during the actual formatting.

The COPY command and standard 40 track, double density, single sided, 5 inch TRSDOS Model III diskettes may be used to transfer TRSDOS Model III diskette files into or out of the NEWDOS/80 system. There are a number of restrictions to this operation.

NEWDOS/80 cannot be used to format a TRSDOS Model III diskette; however, once the user has a formatted empty TRSDOS Model III diskette, he/she may duplicate it repeatedly under NEWDOS/80 using format 5 COPY with the NFMT and BDU options, thus obtaining a stock of formatted, empty TRSDOS Model III diskettes.

The user must assure that where the source and/or destination is a TRSDOS Model III diskette the proper PDRIVE specs are invoked, either implicitly or directly by the SPDN and/or DPDN parameter (see PDRIVE command example 3, section 2.37 for the exact PDRIVE specification).

A file need not previously exist on a TRSDOS Model III diskette in order for it to be copied. NEWDOS/80 will allocate the proper directory entry and diskette space.

Any of COPY formats 1, 2, 3, 4 or 6 may be used to copy files to or from TRSDOS Model III diskettes. Remember, FMT must not be specified. If format 6 is used and one of the source or destination is a TRSDOS Model III diskette, then files marked as SYSTEM files (FPDE 1st byte, bit 6 = 1) are NOT copied.

Files copied between NEWDOS/80 and TRSDOS Model III are always readable though not necessarily usable on the receiving system.

Examples of COPY:

1. COPY XXX:1 YYY:1 In this format 1 COPY, file XXX on the diskette already mounted on drive one is copied as file YYY on that same diskette.

2. COPY,AAA,BBB:2 In this format 1 COPY, the currently mounted diskettes are searched for the file AAA. If found, it is copied as file BBB to the diskette already mounted in drive 2.

3. COPY SUPERZAP/CMD:0 :3 In this format 1 COPY, the file named SUPERZAP/CMD is copied from diskette already mounted in drive 0 to the diskette already mounted in drive 3. Since the file name and name extension are the same for both files, they were dropped from the second file-spec.

4. COPY XXX:1 2 SPDN=9 In this format 1 COPY, SPDN=9 causes, for the duration of the COPY only, all source file I/O to assume that drive 1 has the characteristics specified for drive 9 in the PDRIVE specifications. If we assume that the PDRIVE drive 9 specifications were those for a Model III TRSDOS diskette (see PDRIVE example 3, section 2.37), this COPY will copy file XXX from the TRSDOS Model III diskette already mounted on drive 1 to the NEWDOS/80 diskette already mounted on drive 2.

5. COPY \$XXX:1,YYY:0 In this format 2 COPY, the destination diskette to contain file YYY is not the same diskette as was mounted on drive 0 when the COPY command was initiated. DOS will ask for the mount of the destination and the system diskettes as it needs them.

6. COPY,\$XXX:0 YYY:1 In this format 2 COPY, the source diskette containing file XXX is not the same diskette as was mounted on drive 0 when the COPY command was initiated. DOS will ask for the mount of the source and system diskettes as it needs them.

7. COPY 1 XXX YYY/DAT In this format 3 COPY, the diskette containing file XXX is not the same diskette as the one to contain file YYY/DAT yet both the source and destination diskettes are to use drive 1. DOS will ask for the mount of the source and destination diskettes as it needs them. Note that, as required for format 3 and 4, neither filespec contains a drive number.

8. COPY 0 XXX/DAT /DAT In this format 3 COPY, file XXX/DAT on one diskette is to be copied as file XXX/DAT on another. Both diskettes are to be mounted on drive 0, and DOS will ask for them as needed. Since drive 0 is used and this is format 3 rather than format 4, both the source and destination diskettes must contain NEWDOS/80 systems identical to that mounted on drive 0 when the COPY command was initiated.

9. COPY 0 \$XXX/DAT /DAT This format 4 COPY accomplishes essentially the same thing as the previous example. The difference is that DOS assumes that neither the source nor the destination diskette contains the proper NEWDOS/80 system; so DOS will ask for the mount of the system, source and destination diskettes as it needs them.

10. COPY 0 \$XXX XXX SPDN=9 This format 4 COPY accomplishes the same thing as in example 4 above excepting that only drive 0 is used. For the duration of this COPY, drive 0 uses two sets of PDRIVE specifications. The standard drive 0 specifications are used for the system and destination diskette I/Os, and the system diskette's PDRIVE's drive 9 specifications are used for the source diskette I/Os. Note, in this example, the second filespec was not foreshortened as there was nothing to foreshorten.

11. COPY 0 1 06/01/80 FMT This format 5 COPY is an example of one of the simplest and most commonly used forms of the full diskette COPY. This COPY copies one diskette to another using drive 0 as the source drive and drive 1 as the destination drive. Default track counts for the associated drives are used as diskette track counts. Both drives, other than possibly having different track counts (destination must be greater than or equal to source), have the same characteristics. The operator will be prompted for diskette mounts and error choices, if errors occur. Default parameter Y is in effect, indicating the operator does not care if the destination diskette previously contained data or not. The destination diskette will be formatted before the entire source diskette is copied to it, and it will receive the source diskette's name and password. Its date will be set to 06/01/80. If the destination diskette is to have more tracks than the source, they will be formatted and properly accounted for in the directory such that the destination diskette will be ready for use.

12. COPY 0 1,,NFMT This format 4 COPY is an example of an other form of the simplest and most common full diskette copy. The only difference between this example and the one above is (1) the destination diskette is assumed already formatted, and (2) the current system date will become the destination diskette's date.

13. COPY,0,0,06/01/80,NFMT,USD,KDN,ODN=WATCHDOG,SN=GOODDATA
This format 5 COPY is somewhat the same as the previous example except (1) this is a single drive, two diskette copy, (2) a prompt will be given if the source diskette does not have the name specified, (3) a prompt will be given if the destination diskette does not have the name specified, (4) the destination diskette will retain its old name, (5) it will receive its date from the source diskette. Being a single drive, two diskette copy, more mount prompts will be necessary than for a two drive COPY. Also, because of the large number of diskette mounts that would be involved, this single drive COPY cannot be executed via DOS-CALL (i.e., from BASIC).

14. COPY 0,1,,FMT,CBF This format 6 COPY is an example of one of the simplest and most commonly used forms of multiple file COPY. The destination diskette (to be mounted on drive 1) is to be formatted, and it receives its name and password from the source diskette (to be mounted on drive 0) and its date from the system date. Next, all of the source diskette's files, excepting BOOT/SYS and DIR/SYS, are copied to the destination diskette.

15. COPY 0,1,,NFMT,CBF This format 6 COPY is an example of another of the simplest and mostly commonly used forms of multiple file COPY. The differences between this and example 14 are (1) the destination diskette is not to be formatted, (2) its name, password and date are not changed,

and (3) any source diskette system files (other than BOOT and DIR) that did not already exist on the destination diskette are not copied.

16. COPY 0 1,,NFMT,CBF,USR This format 6 COPY is similar to the previous example except that system and invisible files are not copied.

17. COPY,0,1,,NFMT,CBF,USR,UPD This format 6 COPY is similar to the previous example except that the only source files copied are those marked as updated as well as not being either a system or an invisible file. In this manner, only the files changed since the last backup are backed up now. Remember, COPY does not clear the updated flags on the source diskette; use DOS commands PROT or ATTRIB to do this.

18. COPY,2,3=60,06/01/80,FMT,NDMW,CBF,DDSL=29,DDGA=4
During this format 6 COPY no diskette mount prompts or error choices are to be displayed; the system is to assume the diskettes are already properly mounted. The destination diskette is to be formatted with 60 tracks. The directory will start on lump 29, and will be allocated 4 granules. All source diskette files, except BOOT/SYS and DIR/SYS, will be copied to the destination diskette.

19. COPY 2 3 06/01/80,CBF,CFWO,NFMT For this format 6 COPY, the destination diskette is assumed previously properly formatted and may contain existing files. For each source diskette file, excluding BOOT/SYS and DIR/SYS, the operator will be asked if the file is to be copied to the destination diskette. When all queries are done, the selected files are copied, excepting that system files that did not previously exist on the destination diskette are not copied. If the file already existed on the destination diskette, the file's old data on the destination diskette is lost.

2.15. CREATE Pre-allocate a disk file.

The CREATE command allows a user to create a file and optionally to write to the file a specified number of null records, thereby allocating the file's space as contiguously as possible, given the layout of the free space on the diskette.

There are times when a user program expects one or more of the files it uses to already exist, even though the files may not have any usable data in them; therefore, the user must create the file prior to the program's first use. Also, there are times when the efficiency of a program is reduced if a file's diskette space is scattered all over the diskette; to avoid this, the user should preallocate the needed file space to reduce this scattering.

```
CREATE,filespec1[,LRL=ln1][,REC=count1][,ASE=yn][,ASC=yn]
```

The CREATE DOS command creates new file filespec1 or alters the state of existing file filespec1.

LRL=ln1 specifies the length of each record of the file. ln1 must be a value between 1 and 256; the default value is 256.

REC=count1 specifies the number of records to be initially assigned to a file.

ASE=yn This parameter indicates whether, subsequent to the CREATE command, DOS may automatically allocate more diskette space to this file as necessary. ASE=Y allows this; ASE=N disallows this. The default is ASE=Y.

ASC=yn This parameter indicates whether the DOS close function will be allowed to automatically deallocate excess diskette space. ASC=Y allows this; ASC=N disallows it. The default is ASC=Y.

Enough diskette space is allocated to the file to provide for count1 records each of length ln1. ln1 records of all zeroes are then written to the file, establishing the file EOF at the end of those records. If ASE=N is specified, the file is inhibited against further diskette space allocation, and if ASC=N the file is inhibited against automatic deallocation of excess diskette space.

CREATE command examples:

1. CREATE,XXX:1,LRL=30,REC=2000 File XXX is created, if it did not already exist, on the drive 1 diskette. The record length is 30 and 2000 of these records, containing all 00H bytes, are written to the file. The EOF is left at 60000. Subsequent DOS automatic space allocation and deallocation for this file are allowed.
2. CREATE,YYY:2,200,ASE=N,ASC=N File YYY is created, if it did not already exist, on the drive 2 diskette. The record length is 256 and 200 of these records, containing all 00H bytes, are written to the file. The EOF is left at 51200. Subsequent DOS automatic space allocation and deallocation for this file are not allowed.
3. CREATE,ZZZ:0 File ZZZ is created, if it did not already exist, on the drive 0 diskette. The record length is 256, and the EOF is set to 0. Subsequent DOS automatic space allocation and deallocation for this file are allowed.

2.16. DATE Set computer's current date.

DATE[,mm/dd/yy]

If no parameters are specified, the DATE command displays the current system date in mm/dd/yy format.

If mm/dd/yy is specified, the date mm/dd/yy becomes the system date and is set into the real time clock. mm is the month (value 01 - 12). dd is the day (value 01 - 31). yy is the year (value 00 - 99). No check is made on the validity of the 3 values except to limit them to 2 decimal digits. As the clock reaches 24:00:00, it is reset to 00:00:00 and the date's day within month value is incremented. For the Model I, no adjustment is made for end of month or end of year. For the Model III, end of month and end of year adjustments are done by the ROM.

At reset time, the date is set according to SYSTEM options AY or AZ.

Date command examples:

1. DATE display the system date.
2. DATE,08/1/81 set system date to August 1, 1981.

2.17. DEBUG enable or disable the DEBUG facility.

DEBUG[,yn]

DEBUG or DEBUG,Y DEBUG is enabled (but not entered). This enabling causes a DEBUG entry whenever a user program (such as BASIC, SCRIPSIT, PROFILE, EDIT, etc.) is activated. The DEBUG entry occurs after the program load is completed but just before its first instruction is executed. The purpose of this pre-execution DEBUG entry is to allow the debugging programmer to change the state of a program or its initialization parameters before the program commences execution.

DEBUG,N The above enabling is disabled. At reset/power-on time, DEBUG is disabled.

This command has no effect on the operation of '123' (the simultaneous depressing of the 1, 2 and 3 keys) to enter the DEBUG facility.

Refer to the section 4.1 for the DEBUG facility specifications.

2.18. DIR Display a diskette's directory information.

DIR[:] [dn1] [,A] [,S] [,I] [,U] [,/ext] [,P]

This command displays directory information for the diskette mounted on drive dn1 or if dn1 not specified, on the drive specified by system option AN.

The first display line contains the drive number, the diskette name, its date, the number of tracks, the number of free FDEs and the number of free granules. The values for track count and free granules are based on the current active PDRIVE specification for that drive and if those specifications are not proper, these displayed values may be in error.

The rest of the display contains file information.

If A is not specified, the files are displayed four to a line, giving for each its name and name extension, if any.

If A is specified, DIR will list one file per display line with the display line containing:

1. The file's name.
2. The file's name extension, if any.
3. The file's EOF value in xxx/yyy format where xxx is the relative sector number within the file and yyy is the relative byte number within that sector.
4. The file's logical record size (LRL) in bytes.
5. The number of logical records (RECS) in the file including any partial last record.
6. The number of granules (GRAMS) allocated to the file.
7. The number of diskette space extents (EXT) allocated where that number divided by four and rounded up gives the number of directory entries used by the file.
8. 12 flags providing file information, defined as follows:

1. S = system file.
2. I = invisible file, see ATTRIB DOS command.
3. U = file updated since last time update flags cleared by PROT DOS command.
4. E = file will not be allowed to allocate more space that it already has.
5. C = excess file space beyond EOF is not automatically released during DOS close.
6. - 9. Reserved for future definition.
10. U = non-blank update password exists.
11. A = non-blank access password exists.
12. L = protection Level, see ATTRIB DOS command.

System files are not displayed unless S is specified.

Invisible files are not displayed unless I is specified.

If **U** is specified, only files marked as updated are displayed. Files marked as updated are those files changed via the standard DOS I/O write routine since the last time the update flags were cleared on the target diskette by the PROT or ATTRIB DOS command.

If **/ext** is specified, only those files having the name extension ext are displayed. ext is 0 to 3 characters. Example: DIR,1,/CMD will list all files having extension CMD such as EDTASM/CMD.

If both U and /ext are specified, then only files satisfying both conditions are listed.

When the display screen is full, DIR displays a '?' and waits for the user to respond ENTER to continue or BREAK to terminate the DIR function.

If **P** is specified, the directory information is sent to the printer rather than to the display. Caution, if the printer is not ready, the system will hang waiting for it.

If **\$** is specified, DIR will ask for the mount of the target diskette before the listing and will ask for the remount of the system diskette before exiting. \$ should only be used when drive dnl = 0. There is no provision for changing the PDRIVE specifications internal to the DIR command.

The user must remember that if `dn1` is not specified, the default drive number is that specified by `SYSTEM` option `AN` which is not necessarily 0.

DIR command examples:

1. DIR 0 Display the name and name extension of all non-system, non-invisible files on the diskette currently mounted in drive ~. The files will be listed four per display line.
2. DIR 0,S,I,P Same as the previous example except that system and invisible files are also listed and that the listing is sent to the printer instead of the display.
3. DIR 1,/DAT,U Display the name and name extension of all of the current drive 1 files that are marked as updated and have name extension DAT.
4. DIR 2,A All of drive 2's non-system, non-invisible files are displayed, one file per display line. This display will usually involve more than one display page with the user stepping from one page to the next by pressing ENTER and, if desired; terminating the DIR function by pressing BREAK.
5. DIR \$0 Same as example 1 except the system will ask for the mount of the target diskette on drive 0 and when DIR is done, it will ask for the remount of the system diskette.

2.19. DO Shift to keyboard input from disk.

DO,filespec1[,sectionid]

The DO command executes exactly the same as the DOS command CHAIN (see section 2.9).

2.20. DUMP Dump memory contents to disk.

DUMP,filespec1,start-addr,end-addr[,entry-addr[,relloc-addr]]

The DUMP command writes main memory image data from main memory to the file `filespec1`, starting with the byte at `start-addr` and ending with the byte at `end-addr`.

`Start-addr`, `end-addr`, `entry-addr` and `relloc-addr` are each numeric values less than 65536 decimal or 10000 hex. If the value is hexadecimal, it must be suffixed with a H (i.e. 8000H); otherwise the value is considered decimal. `Start-addr` and `relloc-addr` may be any value 0 - 0FFFFH.

This command operates in two modes, depending on the `entry-addr` value. If the `entry-addr` value = 65535 (0FFFFH), then an exact image of memory is dumped.

The start-value is stored in the file's first 2 bytes, and the rest of the file is the memory dump without any interspersed control bytes. This memory dump file may be displayed or printed via SUPERZAP's DMDB feature, thus allowing debugging to occur later or on another TRS-80 computer.

If entry-addr is less than 65535 (0FFFFH) or is not specified, then the specified area of memory is assumed to be machine executable code and is sent to the file in loader format so that it can be later read back in by the NEWDOS/80 loader, either for execution or simply for load (see LOAD command). If entryaddr is not specified, a value of 402DH (causing return to DOS READY) is used.

CAUTION!! If the user attempts to run or load a file whose start-addr is less than 5200H, DOS will be clobbered.

reloc-addr specifies where the start-addr to end-addr range of bytes is to be loaded to by the LOAD command or when the program file is executed. During write of the object file, the value (reloc-addr) - (start-addr) is added to every load address placed in the object file. This value is also added to the entry-addr if entry-addr is within the start-addr to end-addr range. The actual object code is NOT altered; only the loader control information is.

If filespec1 does not specify a name extension, one is not automatically supplied as is done in TRSDOS.

DUMP command examples:

1. DUMP,PROGRAM/CMD;1,5200H,9ABCH,54EDH dumps the contents of memory from and including 5200H to and including 9ABCH to the file PROGRAM/CMD to exist on drive 1's current diskette. The dump will be in loader format with entry address equal to 54EDH. Subsequently, the file may be loaded back into memory via the DOS command:

LOAD,PROGRAM/CMD

or executed via DOS command:

PROGRAM[,parameters]

2. For this next example, assume that a user program is looping for some reason or has crashed, the personnel to debug the problem are not immediately available, and it is necessary to continue using the computer for other purposes. If a spare formatted diskette is available with sufficient free space, and if 'DFG' can activate MINI-DOS or if the computer is already at DOS READY, then issue the following command:

DUMP,TROUBLE/MEM:2,0,65535,65535

which will dump 65536 bytes of main memory, including ROM, the display, and all of RAM to file TROUBLE/MEM. The first 2 bytes of the file will contain 0000H which is the dump start address; the rest of the file is the memory contents with no interspersed control characters. Once the dump is completed, the operator should set aside the dump diskette for later use by the debugging personnel, optionally press reset, and go on with other tasks. At some later time, debugging personnel can inspect the problem using SUPERZAP's DMDB feature to display or print the contents of file TROUBLE/MEM as if it were actually in memory at the current time.

The debugger must remember that the DOS areas 4000H - 51FFH were altered by DOS actions, including DUMP, after the error occurred and before the dump actually occurred.

2.21. ERROR Display DOS error message.

ERROR,xx

displays the DOS error msg associated with the error number xx where xx is an integer between 0 and 63. Example:

ERROR,24 will display 'FILE NOT IN DIRECTORY'.

2.22. FORMAT Format a diskette for use with the NEWDOS/80 system.

Diskettes as they are received from the manufacturer cannot be used with NEWDOS/80. They must first be magnetically divided into tracks with each track divided into sectors of 256 bytes each. Between 15 and 30 percent of the diskette's bytes are used as format control information and are not available to contain user data.

The DOS command FORMAT does this diskette formatting, setting up the tracks and sectors properly and building the two system files, BOOT/SXS and DIR/SYS, required on every diskette. When done, the diskette is ready to be used as a data diskette with NEWDOS/80.

Formatting can also be done as part of the COPY command, formats 5 and 6 (see section 2.14).

```
FORMAT,dn2[=tc2],name2,mm/dd/yy,password3[,N][,Y][,NDMW]
      [,DDND][,ODN=name1][,KDN][,DDSL=ln1][,DDGA=gc1][,DPDN=dn4]
      [,PFST=tn3[,PFTC=tc3]]
```

FORMAT cannot be executed under MINI-DOS.

In NEWDOS/80 version 2, a track's sectors are read immediately after the track is formatted and before the disk arm is stepped to the next track. Then, after all tracks are formatted, if SYSTEM option BM = Y, the entire diskette is read during the VERIFYING phase. However, if BM=N, this verifying phase is skipped. The user can decide whether or not the verify-at-track format is sufficient and set option BM accordingly.

FORMAT does not allow the user to specify tracks to be locked out, and when an unverifiable sector is encountered, the associated track's lockout byte is not set to FF to indicate lockout. The lockout table is in the standard diskette directory only for compatibility with TRSDOS; NEWDOS/80 does not use it. Remember, NEWDOS/80 does not account for tracks in the directory, it accounts for lumps, which can span tracks. NEWDOS/80 operates under the philosophy, however wrong, that if a diskette cannot be fully formatted it should be discarded.

FORMAT requires all parameters be specified in the command. It does not prompt the user for any.

dn2 is the number of the destination drive to be used during format. Name2 is the name to be assigned to the diskette unless KDN is specified to retain the old name, in which case name2 must still be specified but will be ignored. mm/dd/yy is the date to be assigned to the diskette unless KDD is specified as the diskette date, in which case mm/dd/yy must still be specified but will be ignored. Password3 is the password to be assigned to the diskette. Password3 must conform to the rules for passwords.

Null parameters may be used to invoke default values for diskette name, date and password, using the name NOTNAMED, the system date and the password PASSWORD respectively. Any combination of the 3 null values may be used but where used the null parameters must be delimited by commas, not spaces. See examples 2, 3 and 4 below.

Since FORMAT and COPY share the same NEWDOS/80 code wherever possible, the specifications for the optional parameters are nearly the same as those specified for COPY, formats 5 and 6, the main difference being that only a format is done rather than both a format and a copy. The reader should read the sections for COPY, formats 5 and 6 (see section 2.14) to basically understand FORMAT's optional parameters. Only the differences and two additional options will be given here.

N is the default if neither it nor any of its mutually exclusive keywords are specified.

If =tc2 specified, the diskette will be formatted with tc2 number of tracks; otherwise the diskette will be formatted with the default number of tracks for that drive (see PDRIVE command). If =tc2 value is greater than the number of tracks the drive can handle, format will probably hang trying to step to the non-existent track.

PFST=tn3 and **PFTC=tc3** optional parameters are added to allow the formatting of a range of tracks rather than the entire diskette. If PFST is specified, =tc2 must not be specified, and if PFTC is specified, PFST must be specified. PFST means Partial Format Starting Track, and tn3 specifies the first track to format. If PDRIVE TI flags J or K are applicable for drive dn1, DOS will add one to tn3. PFTC means Partial Format Track Count, and tc3 specifies the number of consecutive ascendingly numbered tracks to format. If PFTC is not specified and PFST is specified, tc3 is assumed equal to 1. After tc3 number of tracks have been formatted and if SYSTEM option BM = Y, the entire diskette will be verified. If this full diskette verify is a problem, cancel the format after verify starts (by pressing up-arrow); remember, each track's sectors were already verified once immediately after the track was formatted.

FORMAT command examples:

1. `FORMAT,0,AAA0,08/01/81,PSWD,Y` The diskette to be mounted, at DOS's request, on drive 0 will be formatted according to the PDRIVE specifications current for that drive. DOS does not care whether the format diskette previously contained data or not. The diskette is named AAA0, dated August 1, 1981, and receives PSWD as its master password.

2. `FORMAT,0,,,Y` This example is identical to the previous example except that default values are used for the diskette name, date and password. The diskette is named `NOTNAMED`, is dated with the current system date and is assigned `PASSWORD` as its password.

3. `FORMAT,1,XXX,,PSWD,N,NDMW,DPDN=4,DDSL=40,DDGA=6` The diskette already mounted on drive 1 must not contain recognizable data. It is formatted according to the system diskette's `PDRIVE` drive 4 specifications (and not according to the existing drive 1 specifications). It is assigned name `XXX` and password `PSWD`; its date is taken from the current system date. The directory starts at the beginning of lump 40 and consists of 6 granules (allows for a maximum of 222 files). Due to `NDMW`, DOS does not ask for the mount of the format diskette nor does it allow error retry.

4. `FORMAT,1,,,Y,PFST=22,PFTC=2` Suppose a power failure destroyed the format of tracks 22 and 23 on a diskette. Using `SUPERZAP`, you have verified that indeed `SECTOR NOT FOUND` error occurs on at least one sector on each of those tracks and, using the `CDS` or `SCOPY` functions of `SUPERZAP`, you have saved in free sectors elsewhere, either on this diskette or another, the readable sectors of those two tracks. Executing this `FORMAT` command will cause only those two tracks to be reformatted; the rest of the information on the diskette is not affected. When done, you can now move back the saved sectors and recreate the ones that were not savable.

2.23. FORMS (Model III only) Set printer parameters.

`FORMS[,WIDTH=xxx][,LINES=yyy]`

The `FORMS` command optionally changes some printer parameters and always lists out the printer parameters.

WIDTH=xxx specifies the number of characters per line where `xxx` must be a value between 9 and 255. If `WIDTH` is not specified, the number of characters per line is not changed.

LINES=xxx specifies the number of lines per page, and must be a value between 1 and 254, where 254 indicates no limit on the lines per page. If `LINES` is not specified, the lines per page value is not changed.

`FORMS` command examples:

1. `FORMS,WIDTH=80,LINES=60` character per line is set to 80 and lines per page to 60.

2. `FORMS,WIDTH=255,LINES=254` Unlimited characters per line and lines per page.

3. `FORMS` Displays current values for characters per line and lines per page.

2.24. FREE Display number of free granules and free FDFs for each diskette currently mounted.

FREE[,P]

For each drive with a diskette mounted, FREE will display the drive number, the diskette name, the diskette date, the number of tracks for the diskette, the number of free FDEs and the number of free granules.

If P is specified, the information will be sent to the printer instead of the display.

FREE command examples:

1. FREE For each diskette currently mount the number of free granules and free directory entries is listed on the display.
2. FREE,P Same as above except the listing is sent to the printer.

2.25. HIMEM Set DOS's high memory value.

HIMEM[,addr1]

DOS maintains a high memory address in the two bytes at Model I location 4049H (Model III location 4411H). This high memory value is used by COPY, BASIC, EDTASM, DISASSEM and LMOFFSET as the upper limit of the memory they can use. User programs should also use this 2 byte HIMEM value as their upper limits. Caution! The loader does not use HIMEM as its upper limit during program load.

If no parameters are specified, the HIMEM command displays in hexadecimal the current high memory value.

If addr1 is specified, the DOS high memory address is set to addr1 which must be an integer between 28672 and 65535 decimal (7000H - 0FFFFH hexadecimal).

HIMEM command examples:

1. HIMEM Displays the current DOS high memory address.
2. HIMEM,49000 Set DOS's high memory value to 49000 (0BF68H).

2.26. JKL Send the current contents of the display to the printer.

JKL has no parameters. This command uses the same routine used by the 'JKL' triple key function (see section 4.5). JKL simply dumps the display contents to the printer. If system option AK=Y, hex codes >= 80H (which includes the graphics) will be transmitted unchanged; otherwise a period will be

substituted for them. Hex codes < 20H will be displayed as periods. Pressing BREAK during JKL print will terminate the JKL function.

JKL's main use will be either via CMD"JKL" from BASIC or via DOS-CALL from a user program.

2.27. KILL delete a file.

This command deletes a file from a diskette. The file is no longer accessible by normal methods and is no longer known to DOS.

`KILL,filespec1`

The file `filespec1` is deleted from the current diskette mounted on the specified drive. If a drive number was not specified, then all mounted diskettes are searched, starting with the diskette on drive 0, and the delete is done on the 1st file found having the specified name and name extension.

KILL action is as follows:

1. If the file was allocated file space on the diskette, the space is released, and becomes available for subsequent assignment to other files. The file's data, if any, on the diskette is not altered by the KILL. This data, though no longer accessible, is not written over until the associated file space is reassigned to another file and those sectors actually written to.
2. The file's FPDE and any owned FXDEs are freed by zeroing bit 4 of the 1st byte of each and by zeroing the associated HIT sector byte for each. Except for that bit 4, none of the associated FPDE and FXDEs are altered by normal DOS operation until that FDE is reassigned to another file by DOS.

If the user has inadvertently killed a file that shouldn't have been, since neither the associated FDE's or the diskette space used by the file is changed by DOS until DOS has a need to, it is possible to reconstruct the FPDE and FXDEs and reallocate the space. To do this, you must be extremely familiar with the workings of the directories; do not call Apparatus as this is a major undertaking and not something that can be quickly taught. If you don't know how to do it, forget it!

If you have more than a few files to delete at one time from a diskette, use the PURGE command.

KILL command examples:

1. `KILL XXX/BAS:1` The file XXX/BAS on the diskette mounted on drive 1 is killed.
2. `KILL YYY` Starting with drive 0, mounted diskettes are searched until file YYY is found on one of them. That file is then killed. If other mounted diskettes also contain a YYY file, the other YYY files are not killed.

2.28. LC Set keyboard a - z toggle switch to the specified state.

LC[,yn]

LC or LC,Y sets the keyboard lower case a - z toggle switch to accept a - z without change.

LC,N sets the keyboard lower case a - z toggle switch to change lower case a - z to upper case A - Z.

For the Model I, the LC command has no effect unless the lower case driver is active (see LCDVR command).

2.29. LCDVR (Model I only) Lower case driver.

LCDVR[,x[,s]]

In NEWDOS/80 version 1, the lower case driver that processed keyboard lower case alphabets and which sent lower case displayed characters to the display was a separate program that executed from high memory. In version 2, the lower case driver is an integral part of the Model I NEWDOS/80.

If x = Y, the lower case driver routine is activated, and if x = N, the routine is deactivated. When the lower case driver routine is active:

1. Keyboard input a - z characters are processed according to the a - z toggle switch.
2. ASCII codes 96 - 127 (60H - 7FH) are displayed as their proper characters and are not changed to 64 - 95 (40H - 5FH) by the ROM display routine.

The second parameter is meaningful only when x = Y, performs the same as the first parameter of LC command, initially setting the a - z toggle switch to accept a - z (if s = Y) or convert a - z to A - Z (if s = N).

Once the lower case driver is activated, pressing shift 0 will switch the driver back and forth between accepting lower case letters and converting lower case letters to upper case. Further, DOS command LC may be used to explicitly set one or the other of those states.

To use the lower case driver, NEWDOS/80's keyboard and display intercept routines must be enabled. Other routines (excluding ROUTE) that disable these NEWDOS/80 functions will also disable the lower case driver (one example is using the circular buffer in the spooler).

If no parameters are specified, the command is assumed to be LCDVR,Y,N.

This lower case driver operates somewhat differently than the LCDVR program supplied with Version 1. In Version 1, if lower case a - z was being converted to upper case A - Z, then upper case A - Z was also being converted to lower case a - z. Version 2 does not convert upper case A - Z to lower case a - z; instead a true capital letter lock is done:

LCDVR command examples:

1. LCDVR The lower case driver routine is activated and the lower case switch is set to convert lower case a - z to upper case A - Z.
2. LCDVR,Y,Y The lower case driver routine is activated, and the lower case switch is set to accept lower case a - z without modification.
3. LCDVR,N The lower case driver routine is deactivated.

2.30. LIB Display NEWDOS/80 library commands.

LIB requires no parameters. It displays the library commands of NEWDOS/80. Commands FORMAT, COPY and APPEND execute in memory 5200H and up, and, along with CHAIN, cannot be executed in MINI-DOS. The other commands execute from the DOS overlay area, 4D00H-51FFH, and, except for CHAIN, can be executed under MINI-DOS.

2.31. LIST List a text file on the display.

`LIST,filespec1[,start-line[,line-count]]`

This command sends the contents of file `filespec1` to the display. Though file `filespec1` need not be a text file, if it is not, the resulting display will not be very meaningful. Examples of text files are BASIC programs saved with the A option, BASIC files written using PRINT, assembler, FORTRAN and COBOL source text files, SCRIPSIT files saved with the A option and Electric Pencil files. To list a non-text file, use SUPERZAP.

No check is made on the character representations except to modulate characters whose hexadecimal values are between 80H and FFH into the range 00H to 7FH and to replace with a period all characters whose hexadecimal value is less than 20H or greater than the high ASCII character value specified by the SYSTEM option AX.

If `start-line` (decimal value 1 - 65535) is specified, listing will start with that line where a line is considered to end with the ENTER or EOL character 0DH.

If `line-count` is specified, then the number of lines displayed is limited to either `line-count` or the number of lines in the file from the start point, whichever is less. If `line-count` is specified, `start-line` must also be specified.

Pressing right arrow will cause a display pause when hex char 0DH is encountered or after 256 bytes have been displayed, whichever comes first. Pressing ENTER will continue the displaying. Pressing up-arrow will terminate LIST.

Aside from just listing a file, LIST is useful where text files maintain a date/time stamp near the beginning. If the user has multiple copies of a text

file, it may be necessary to look at the file beginning to determine which copy is the most recent.

LIST command examples:

1. LIST,BASEPROG/BAS displays the entire contents of file BASEPROG/BAS.
2. LIST,XXX,1,6 displays the first 6 lines of file XXX.
3. LIST,YYY:1,200 displays the contents of file YYY from the 200th line to the end of the file.

2.32. LOAD Load a Z-80 machine language file into RAM.

LOAD,filespec1

This command loads the Z-80 machine language file filespec1 into RAM, and stores its entry address into the two bytes at 4403H (17411 decimal). The file must be in proper loader format, such as created by DUMP or EDTASM. The load proceeds using control data from the file. If the file loads over any part of the resident DOS (4000H - 4CFFH) or its overlay area (4D00H - 51FFH), serious and maybe file damaging trouble will occur; with luck, the system will hang.

LOAD is used when a program or data is to be loaded into RAM for later use by other programs. An example is loading programs, which will be invoked via BASIC's USR function. Remember, the entry address is stored in the two bytes at 4403H (17411 decimal); this is not done in TRSDOS.

LOAD command examples:

1. LOAD,OVERLAY/OBJ:1 The object code module OVERLAY/OBJ is loaded into main memory from the diskette mounted on drive 1. The load control information within file OVERLAY/OBJ determines what is to be loaded and where in main memory it is to be loaded.
2. Suppose that BASIC does not use all of high memory and that a BASIC program wishes to load the program USR3PGM/OBJ into high memory and later execute it as the BASIC USR3 function. Executing the BASIC statements:

```
CMD"LOAD,USR3PGM/OBJ"
DEFUSR3 = (PEEK(17411) + 256 * PEEK(17412) - 65536
```

will set this up.

2.33. MDBORT Terminate MINI-DOS and go to DOS READY.

MDBORT has no parameters. It should only be executed when NEWDOS/80 is in MINI-DOS state. MINI-DOS state is terminated, the pre-MINI-DOS state purged and the system goes to DOS READY.

The purpose of MDBORT is to provide for the situation where the operator does not want to continue the main program which was interrupted by the simultaneous depression of the D, F and G keys (which invoked MINI-DOS).

2.34. MDCOPY Copy a file while under MINI-DOS.

```
MDCOPY,filespec1[,TO],filespec2
```

The regular COPY command cannot be executed under MINI-DOS. MDCOPY gives the user a restricted and quite slow form of file copy, which does execute under MINI-DOS.

MDCOPY copies the contents of file filespec1 to the new or existing file filespec2. File filespec1 is not altered, and the previous contents of file filespec2, if any, are lost. Filespec2 may not be foreshortened as is allowed for COPY.

MDCOPY command example:

```
MDCOPY XXX/DAT:0 YYY/DAT:1
```

The contents of file XXX/DAT on the diskette currently mounted on drive is copied as file YYY/DAT onto the diskette currently mounted on drive 1.

2.35. MDRET Exit from MINI-DOS and return to main program.

MDRET has no parameters. The system exits MINI-DOS state and continues the main program at the point where it was interrupted by the invocation of MINK DOS (simultaneous depression of the D, F and G keys). If the cursor was displayed before 'DFG', it will be redisplayed. If the 'DFG' interruption occurred while the key input buffer contained a partial input record, that partial record is still there even though it is no longer displayed. The user should continue keying exactly where he/she left off.

If the invocation of MINI-DOS occurred during the timer interrupt rather than the key intercept, one or more of D, F or G may appear as spurious input keys after MDRET is executed. The user should backspace over them. The user and DOS have no control over these spurious input chars; therefore DFG should not be pressed when a program is in text overwrite mode, such as SCRIPSIT or Electric Pencil; instead go into command mode where the spurious characters can be backspaced over without damage to the text.

2.36. PAUSE Display message and pause waiting on ENTER.

PAUSE,msg

The message msg is not redisplayed if the PAUSE command itself was displayed. If the PAUSE command was not displayed, as occurs if it is executed under DOS-CALL, the message msg is displayed. In any event, the message PRESS "ENTER" WHEN READY TO CONTINUE is displayed on the next line. DOS then waits for the user to press the ENTER key. The PAUSE command is one of the four ways of causing a pause in chaining, and can also be used when a series of commands in main memory are being executed by a series of DOS-CALLS.

PAUSE command example:

PAUSE,MOUNT DISKETTE LABELED "PRIMARY" ON DRIVE 1.

This message will appear on the display and will be followed on the next , display line by the message PRESS "ENTER" WHEN READY TO CONTINUE. DOS waits for the user to press ENTER which presumably he/she will do after the proper diskette has been mounted in drive 1. DOS doesn't check to see if the user has done what was requested; all DOS does is wait for the ENTER.

2.37. PDRIVE Assign default attributes to a physical drive.

PDRIVE[,password1:]dn1,[dn2[=dn3]][,TI=type1][,TD=type2][,TC=tc1]
[,SPT=scq][,TSR=rc1][,GPL=gc2][,DDSL=ln1][,DDGA=gc1][,A]

NEWDOS/80 has limited capabilities for operating with a mixture of 5 inch disk drives and to a lesser extent 8 inch disk drives. PDRIVE is the command method used to inform NEWDOS/80 of a particular physical drive's characteristics.

Each PDRIVE command lists the resulting specifications for 10 drives even though the actual number of drives eligible for I/O is limited by the SYSTEM option AL and in no case exceeds 4. Those drives within the range of SYSTEM option AL are flagged on the PDRIVE display by an asterisk suffixed to the drive number. The specifications for the 10 drives is maintained on the system diskette mounted on drive dn1. For efficiency reasons, DOS normally uses drive specifications from a table it has in main memory. This main memory PDRIVE table contains specifications for 1 to 4 drives, depending upon the SYSTEM option AL value, and is automatically reloaded from the drive 0 diskette at power on and reset if and only if the specifications for all 10 drives are error free (otherwise the reset hangs). This table is also immediately reloaded by a PDRIVE command specifying the A parameter (see below).

Drive dn1 is the drive containing the system diskette whose control information (in the 3rd sector) is being updated. Drive dn2 indicates which physical drive of the 10 represented in the control information sector on drive dn1 is having its control information updated.

For example, if the PDRIVE command is PDRIVE,1,4,TC=80 then the diskette on drive 1 is read to obtain the PDRIVE control information and is updated to contain the new drive 4 specification. Drive 1's PDRIVE control information contains the specifications for ten drives, dn2 values 0 through 9, and it is the fifth drive's information (for dn2 = 4) that is changed. The specifications for the other nine drives are not changed.

If passwords are enabled, then password1 must be specified and be the master password for the diskette on drive dn1. Otherwise, password1 may be left out of the command.

Control data is changed only for the parameters specified; parameters not specified are not changed. If any errors are displayed, the dn1 diskette must NOT be used as the system diskette during a reset/power-on until the errors are corrected.

PDRIVE,dn1 will list the 10 PDRIVE specifications contained in the control data on the system diskette mounted on drive dn1.

dn2 must be specified if any other optional parameters except A are specified. If dn2 is specified, it must be the 1st parameter following dn1.

dn2=dn3 causes drive dn2 to assume the PDRIVE specifications of drive dn3. This is done before any other optional parameters are interpreted.

TI=type1 specifies the type of disk drive interface. type1 consists of one or more alphabetic letter flags chosen from the list below. For the Model I, one and only one of flags A, B, C or E must be chosen. For the Model III, one and only one of flags A or D must be chosen. The other flags are optional depending upon the interface. Certain flags are inter-drive mutually exclusive meaning that for a given drive dn1, if one dn2 drive specifies a flag that is interdrive mutually exclusive with another flag, then another dn2 drive may not specify the excluded flag. For now, flags B, C and E are interdrive mutually exclusive for the Model I.

Flag **A** means the standard disk interface is to be used for diskette I/O for this drive. For the Model I this interface supports drive types A and C. For the Model III this interface supports drive types A, C, E and G.

Flag **B** (Model I only) means that an OMIKRON mapper type interface is installed and is to be used for I/O for this drive. This interface supports drive types A, B, C and D.

Flag **C** (Model I only) means that a PERCOM doubler type interface is installed and is to be used for I/O for this drive. This interface supports drive types A, C, E and G.

Flag **D** (Model III only) means that an Apparat disk controller type interface is installed and is to be used for I/O for this drive. This interface supports drive types A through H (drive types F and H require a Model III speed up modification).

Flag **E** (Model I only) means that an LNW type interface is installed and is to be used for I/O for this drive. This interface supports drive types A through H.

Flag **H** means head settle delay is to be done whenever DOS changes from another drive to this drive. For Model I and Model III 5 inch drives, the heads for all 5 inch drives are loaded when the motors go on, and this extra time delay is NOT needed. Flag H is needed for 8" drives.

Flag **I** means the lowest numbered sector on each track is sector 1. This is the normal state for Model III TRSDOS diskettes. If flag I is not specified, the lowest numbered sector on each track is assumed to be 0, which is the state for the Model I and for NEWDOS/80 on the Model III.

Flag **J** means the track numbers start from 1. If flag J is not specified, track numbers are assumed to start from 0, which is the standard state for the Model I and the Model III.

Flag **K** means track 0 is formatted (or is to be formatted) in density opposite to that of the diskette's other tracks. This makes track 0 unavailable for normal I/O. Flag J is implicitly set by flag K. The purpose of formatting track 0 in opposite density is to allow a double density (Model I) or single density (Model III) SYSTEM diskette to be booted up. The Model I ROM must be able to read the boot sector in single density, and the Model III ROM must be able to read the boot sector in double density. Setting flag K causes FORMAT and COPY with format to format track 0 in the opposite density and to store the required boot sector onto that track for the ROMs to use. With flag K set, normal DOS I/O to track actually goes to track 1, 1 to 2, etc. Flag K must be specified for a drive that is to read a double density diskette created by the PERCOM type doubler interface under NEWDOS/80 version 1 or any other DOS except NEWDOS/80 version 2 or higher. For NEWDOS/80 version 2 Model I, double density data diskettes do not have to reserve track 0 for opposite density if those diskettes will never be used on a drive 0 whose PDRIVE specifies double density. Flag K must NOT be specified for standard Model III diskettes, unless for some reason the user wants a single density system diskette on the Model III or is making a double density diskette to be read on the Model I that does not have NEWDOS/80 version 2. When flag K is specified, then TC must specify one less track than would be specified if flag were not specified. Further, due to the differing sequence in which consecutive sectors are stored on the diskettes, double sided, double density diskettes created under the patched NEWDOS/80 version 1 are not readable under NEWDOS/80 version 2. To transfer files on those diskettes to Version 2, they must first be moved (using Version 1) to either single sided (either density) or double sided, single density diskettes.

Flag **L** means two step pulses between tracks. This allows a 35 or 40 track diskette to be read on an 80 track drive. Writing can also be done in this manner, but the 35 or 40 track drives have trouble reading some of the sectors so writing is not recommended.

Flag **M** means the diskettes are standard TRSDOS Model III diskettes. Flag M implies flag I. The COPY DOS command is the only function within NEWDOS/80 that will honor or even notice a TRSDOS Model III diskette as distinct from a NEWDOS/80 diskette, and even this will not occur unless flag M is set.

Flags F through G and N through Z are reserved for future definition.

TD is the Type of Drive specification. The definitions are:

1. TD=A 5 inch, single density, single sided drive.
2. TD=B 8 inch, single density, single sided drive.
3. TD=C 5 inch, single density, double sided drive.
4. TD=D 8 inch, single density, double sided drive.
5. TD=E 5 inch, double density, single sided drive.
6. TD=F 8 inch, double density, single sided drive.
7. TD=G 5 inch, double density, double sided drive.
8. TD=H 8 inch, double density, double sided drive.

If a CPU speed up module is installed in the computer that reverts to normal CPU during disk I/O, this reversion must not slow the CPU speed to less than the original rated CPU speed for that model. NEWDOS/80's disk I/O loops, especially for the Model 1 for drive types B, D, E and G, cannot tolerate any reduced CPU speed below the original speed. In limited testing and with SYSTEM option BJ properly set, NEWDOS/80 Version 2 has run disk I/O successfully without the need to turn off the CPU speed; however, Apparat does not guarantee such performance.

TD=F and TD=H require a CPU speed up module installed in the computer which at least doubles the CPU's speed during disk I/O.

For drive types C, D, G and H, the current NEWDOS/80 interfaces (TI flags A, B, C, D or E) consider a double sided diskette as a single volume (i.e., only one directory) with each track having its lower numbered sectors on the first side and the higher numbered sectors on the second side. Pin 32 is used to select the 2nd side (special cables required), and any drive on the cable that shunts pin 32 over as a drive 3 select must have that shunt wire cut to prevent that drive from being selected when another drive's 2nd side is being selected. Double sided, double density 40 and 80 track drives have been used on the Models I and III under NEWDOS/80 Version 2.

One of the reasons Apparat never supported double density in Version 1 was that most drives did not work reliably in double density. Whether this was the fault of the drives, the diskettes, the data separator or the controller was never really ascertained. Over the last nine months, things have improved somewhat, but double density is still not as reliable as single density and probably never will be. Apparat was informed that the two byte pattern 6DB6 is a much better "worst case" double density pattern than the E5's used in single density, and indeed the 6DB6 pattern is such. In fact, it is such a good "worst case" condition that a good percentage of certified double sided, double density diskettes will fail format. To many users, this will prove intolerable and they will want to apply the ZAP that goes back to the E5 pattern, if it is not already applied. However, using the E5 pattern in double density means that the user will increase the probability that a diskette that formats successfully will at some future time fail.

TC=tc1 specifies the number of tracks on the disk, excluding track 0 if TI flag K is set. If flag K is not set, TC=35 for a 35 track drive, TC=40 for a 40 track, etc. If flag K is set, then TC=34 for a 35 track drive, TC=39 for a 40 track, etc.

SPT=scl specifies the number of sectors per track. For double sided, single volume diskettes (TD = C, D, G or H), scl must be twice what it would be if single sided diskettes. scl may be any value from 1 to the maximum number of 256 bytes sectors the track can physically hold. For each of the above specified drive types, the maximum number of sectors per track is: A=10, B=17, C=20, D=34, E=18, F=26, G=36 and H=52.

TSR=rcl specifies the track stepping pulse time code the controller uses for this drive. rcl is a value from 0 to 3 and becomes part of the SEEK, STEP and RESTORE commands sent to the controller. For the Model I and III standard controllers, TSR=0 gives 5 ms stepping, TSR=1 gives 10ms stepping, TSR=2 gives 20ms stepping and TSR=3 gives 40ms stepping. TSR=3 was the original standard for the Model I, with some users using TSR=2 or TSR=1 for certain drives. The Model III appears to use TSR=0 as standard. If you are having drive trouble, the safest setting is TSR=3 (fastest stepping rate for the Model I is 12ms).

GPL=gc2 specifies the number of granules per lump where gc2 is a value between 2 and 8. In TRSDOS for the Model I and III and the older versions of NEWDOS, disk space allocation was done via granules (5 sectors per granule on the Model I and 3 per granule on the Model III) and tracks (2 granules per track on the Model I and 6 granules per track on the Model III). In NEWDOS/80 version 2, for both the Models I and III, there are still 5 sectors per granule, and 2 to 8 granules per lump (not track). Wherever a track number appeared in the directory (in the GAT sector and in the FDE two byte extent elements), it has been replaced with a lump number. Doing so allows a granule to start in one track and end in another and allows double density and 8 inch diskettes to maximize the number of sectors per track while keeping the same directory format. GPL=2 maintains compatibility with the old 35 track single density diskettes, as the directories will be exactly the same and transferable back and forth between the Model I TRSDOS and NEWDOS versions before NEWDOS/80 version 2. However, by going to GPL=8 the directory can now accommodate $192 \times 8 \times 5 = 7680$ sectors or 1,966,000 bytes.

DDSL=ln1 is the logical equivalent of and replacement for the DDST parameter used in NEWDOS/80 version 1. IS specifies the number of the lump at whose first sector is to contain the directory's 1st sector. This value is stored in the boot sector 3rd byte during diskette format and is used when necessary to find the directory. It is also used during diskette format to determine where to put the directory. In the older systems, the 3rd byte of the boot sector contained the track number in whose 1st sector the directory started. Since tracks are not used in space allocation and control in NEWDOS/80 version 2, the 3rd byte of the boot now contains the number of the lump in whose 1st sector the directory starts. To determine the relative sector number of the directory's 1st sector (the GAT sector), access the boot sector's 3rd byte and multiply that value by 5 times GPL. DDSL=17 maintains compatibility with the standard 35 track, single sided, single density diskettes. DDSL should be set to the value used for the DDST parameter in NEWDOS/80 version 1.

DDGA=gcl specifies the default number of granules to be allocated to the directory when it is created during format, where gcl is a value between 2 and 6. DDGA=2 should be specified for standard 35 track, single density, single sided compatibility. $gcl > 2$ allows the user to have more than 62 files on a data diskette with the maximum being 222 files.

A specifies that if and only if no errors were found during the checking of the specifications for all the drives, then the specifications for SYSTEM option AL number of drives is loaded into the main memory PDRIVE table to immediately become the controlling data for those drives; this eliminates the need for a reset. If parameter A is specified, dn1 must = 0.

PDRIVE is executable under MINI-DOS.

PDRIVE command examples:

1. PDRIVE,dn1,dn2,TI=A,TD=A,TC=35,SPT=10,TSR=3,GPL=2,DDSL=17,DDGA=2
is the PDRIVE specification for a standard 5 inch, 35 track, single density, single sided diskette used for communication in the Model I world. This specification can also be used on the Model III to read the diskette providing the directory address marks are correct (see SYSTEM option AN).

2. PDRIVE,dn1,dn2,TI=A,TD=E,TC=40,SPT=18,TSR=3,GPL=2,DDSL=17,DDGA=2
is the Model III specification (Model I, use TI=C) for a standard 5 inch, 40 track, double density, single sided diskette used for communication through out the NEWDOS/80 Model III world. Using this specification, this diskette can also be read on the Model I in a drive other than 0 if a double density modification is installed in the expansion interface.

3. PDRIVE,dn1,dn2,TI=AM,TD=E,TC=40,SPT=18,TSR=3,GPL=6,DDSL=17,DDGA=2
is the Model III specification (Model I, use TI=CM or EM) for reading or writing to a TRSDOS Model III standard 5 inch, double density, single sided diskette. A 40 track, double density, single sided 5 inch diskette is the only type TRSDOS Model III diskette that NEWDOS/80 can handle. GPL=6 is mandatory. Since a TRSDOS Model III diskette cannot be formatted by NEWDOS/80, DDSL and DDGA are meaningless. In NEWDOS/80 (double density mod must be installed for Model I), only the COPY DOS command can be used with TRSDOS Model III diskettes excepting that diskette sectors can be read/written via SUPERZAP by using the DD, DM, DTS, VDS, CDS, CDD, etc. functions that do not refer to files (i.e., don't use DFS).

4. PDRIVE,dn1,dn2,TI=A,TD=C,TC=80,SPT=20,TSR=2,GPL=8,DDSL=20,DDGA=6
is the specification for a 5 inch, 80 track, single density, double sided, single volume diskette with 20ms stepping, 8 granules per lump, with the directory positioned at the diskette halfway point and maximum size directory. For the Model III, the single density drive 0 restriction applies.

5. PDRIVE,dnq,dn2,TI=A,TD=G,TC=80,SPT=36,TSR=2,GPL=8,DDSL=35,DDGA=6
is the Model III specification (Model I, use TI=C or E) for a 5 inch, 80 track, double density, double sided, single volume diskette to use 20ms stepping, 8 granules per lump, maximum size directory positioned at the diskette halfway point. For the Model I, the double density drive restriction applies.

6. PDRIVE,dn1,dn2,TI=CK,TD=E,TC=39,SPT=18,TSR=3,GPL=2,DDSL=17,DDGA=2
is the Model I specification (Model III, use TI=AK) for 5 inch, 40 track, double density, single sided diskette that has track 0 formatted in single density, hence only 39 tracks available for regular use. This specification will handle double density diskettes formatted by TRSDOS and NEWDOS/80 version 1 running under the PERCOM doubler. This specification will also be used when generating a double density diskette

to be the system diskette in drive 0 for the Model I. For LNW Model I interface, use TI=EK.

7. PDRIVE,dn1,dn2,TI=CK,TD=G,TC=79,SPT=36,TSR=3;GFL=8,DDSL=35,DDGA=6 is the Model I specification (Model III, use TI=AK) for a 5 inch, 80 track, double density, double sided, single volume diskette that has track formatted single density. For the LNW Model I interface, use TI=EK.

Warning!!! Double sided, double density diskettes used on the patched NEWDOS/80, version 1 are not usable on Version 2 (see TI flag K discussion).

8. PDRIVE,dn1,dn2,TI=AL,TD=A,TC=35,SPT=10,TSR=3,GPL=2,DDSL=17,DDGA=2 is the specification for a 5 inch, 35 track, single sided, single density diskette that is to be read on an 80 track drive. The 80 track drives step only half as far as the 35 and 40s for each data track; setting flag L causes 2 steps to be taken for each data track stepped.

9. PDRIVE,dn1,dn2,TI=BH,TD=B,TC=77,SPT=17,TSR=3,GPL=3,DDSL=17,DDGA=6 is the Model I specification for an 8 inch, 77 track, single sided, single density diskette. Note, NEWDOS/80 version 1 used SPT=15 and an implied GPL=3, and to read those diskettes, SPT=15 and GPL=3 must be used. It is recommended that a COPY be done to convert those diskettes to SPT=17, thus gaining 12% more diskette space. Flag H causes head load settle delay to be used, required for most 8 inch drives.

10. PDRIVE,dn1,dn2,TI=BH,TD=D,TC=77,SPT=34,TSR=3,GPL=8,DDSL=17,DDGA=6 is the Model I specification for an 8 inch, 77 track, single density, double sided, single volume diskette with head load settle delay required.

11. PDRIVE,dn1,dn2=dn3 is the specification to cause drive dn2 to receive as its specifications those of drive dn3.

12. PDRIVE,dn1,dn2=dn3,TC=40,TSR=2 is the specification to cause drive dn2 to receive as its specifications those of drive dn3 and then to apply new values for TC and TSR.

13. PDRIVE,0,A causes the PDRIVE data for SYSTEM option AL number of drives to be loaded into the main memory PDRIVE table if and only if the full display of the specifications shows no error.

14. PDRIVE,0,dn2=dn3,A changes drive 0's specifications for dn2 to be those of dn3, and then performs as in the above example.

2.38. PRINT List a text file on the printer.

PRINT,filespec1[,start-line[,line-count]]

PRINT executes identical to LIST, excepting the listing goes to the printer instead of the display. Refer to DOS command LIST for specifications and examples.

2.39. PROT Alter some diskette control data.

```
PROT,[password1:]dn1[,NAME=name1][,DATE=mm/dd/yy][,RUF]
    [,PW=password2][,LOCK][,UNLOCK]
```

At least one optional parameter must be specified. The target diskette is mounted on drive dn1. If passwords are enabled, password1 must be specified and must equal the diskette's master password.

NAME=name1 The diskette is given the name name1.

DATE=mm/dd/yy The diskette is given the date mm/dd/yy.

RUF Reset Updated Flags. This option turns off the updated flags for all files on the diskette. If a user backs up only those files having the updated flag on (see UPD option of COPY) off, executing PROT with the RUF option after the copying is completed turns off the updated flags so the files will not be eligible for a subsequent backup until the file is subsequently updated. Simply writing or rewriting one sector of the file, whether or not anything was actually changed, causes DOS to turn on a file's updated flag.

PW=password2 Password2 must conform to the rules for passwords, with null set as all blanks. The diskette receives password2 as its password.

LOCK All files of the diskette, except system and invisible files, are given the diskette master password as both their access and update passwords. If password2 specified, it is used. This feature used to be the only way a user, in a password enabled system, could get to a file whose password(s) he/she had forgotten, if the user did know the diskette master password. It has the unfortunate drawback in that it changes the passwords for all, except system and invisible, files on the diskette; thus causing the user to reassign passwords to all the others as well as to the file whose passwords he/she forgot. An easier way is available if the user knows the password of at least one NEWDOS/80 system diskette or better still, has a NEWDOS/80 system diskette with passwords disabled (system option AA = N). With passwords disabled, the user can use ATTRIB to, directly reassign new passwords to the file whose passwords are forgotten without having to affect other user files on the diskette. Then passwords can be re-enabled.

UNLOCK The access and update passwords of all of the diskette's files, except system and invisible files, are set to all blanks, meaning no passwords for those files.

PROT command examples:

1. PROT,2,RUF The updated flag is cleared for each file on the diskette currently mounted on drive 2.

2. PROT,OLDPSWD:1,NAME=AAB3,DATE=07/15/81,PW=NEWPSWD
In this example, passwords are enabled; therefore the diskette's master password OLDPSWD was required. The diskette control information for the diskette mounted on drive 1 is changed such that its name is AAB3, its date is July 15, 1981 and its new master password is NEWPSWD.

2.40. PURGE Selectively kill files from a diskette.

PURGE,[password1:]dn1[,/ext][,USR]

The diskette mounted on drive dn1 is used for this command. If passwords are enabled" password1 must be specified and must be equal to the diskette's master password.

For each file, except BOOT/SYS and DIR/SYS, on the diskette, DOS asks the operator if the file is to be killed. If the file is to be killed, respond Y; the file will be immediately killed, as if a KILL command has been issued. If the file is NOT to be killed, respond N. Respond Q if you wish to quit the PURGE function.

/ext If this option is specified, the purge queries are limited to only those files having name extension ext where ext is 0 to 3 characters.

USR If this option is specified, system and invisible files are not included in the PURGE function.

PURGE command examples:

1. PURGE,1 For each file, except BOOT/SYS and DIR/SYS, on the diskette currently mounted on drive 1, DOS asks if the file is to be killed. If the response is Y, the file is killed.
2. PURGE,0,/DAT For each file on the diskette currently mounted on drive 0 that has name extension DAT, DOS asks if the file is to be killed and does so if the response is Y.
3. PURGE,0,USR For each non-system, non-invisible file on the diskette currently mounted on drive 0, DOS asks if the file is to be killed and does so if the response is Y.

2.41. R Repeat the previous DOS command.

This command causes the re-execution of the previous DOS command, excluding the command R. Example:

DIR 1 followed by:
R

will execute the same as if the two DOS commands had been:

DIR 1
DIR 1

The R command can not be executed from BASIC via CMD"doscmd" as that function requires that the command, excluding ENTER, must be 2 or more characters long.

The R command has no parameters and must be keyed exactly as R followed by ENTER. If more than 2 characters are keyed into the buffer and then

backspaced so that DOS only sees the R and the ENTER, the previous DOS command that was residing in the command buffer will still have been altered and the R command will either fail or in rare circumstances, execute something different than what the operator expected.

If the previous DOS command is no longer intact in the DOS command buffer, the results of the R command are unpredictable.

If SYSTEM option BE = N, the R command does not execute the previous DOS command but instead simply returns to DOS READY.

2.42. RENAME Rename a file.

```
RENAME,filespec1[,TO],filespec2
```

The file filespec1 is renamed to filespec2, where filespec2 consists of only a name and optionally a name extension. If filespec1 does not specify a drive number, then all mounted diskettes are searched, and the first file encountered matching filespec1's name and name extension is renamed. RENAME change only the file's name and name extension; nothing else is changed.

RENAME command example:

```
RENAME XXX/DAT:1 YYY/OBJ The file XXX/DAT on the diskette currently
mounted on drive 1 has its name changed to YYY and its extension changed
to OBJ.
```

2.43. ROUTE

1. ROUTE
2. ROUTE,CLEAR
3. RDUTE,dev1[,dev2][,dev3]....

The purpose of the ROUTE command is to allow some flexibility from where the keyboard and/or RS-232 input is received and to where display, printer and RS-232 output is sent.

At the conclusion of a ROUTE command, any existing routes are displayed; if none, nothing is displayed. ROUTE with no parameters does nothing except display the existing routes.

```
ROUTE,CLEAR          clears all routes.
```

dev1 specifies the device being routed. dev2, dev3, etc. specify the device(s) being routed to (the routed-to devices) when dev1 is an output device or routed from (the routed-from devices) when dev1 is an input device. For the Model I, the device codes are KB for the keyboard, DO for the display PR for the printer and NL for null (meaning nothing is transferred). For the Model III, RI for the RS-232 input and RO for the RS-232 output are added to the above 3 codes. An input device (KB or RI) may not be routed to an output

device (DO, PR or RO), and an output device may not be routed to an input device.

Whenever dev1 is specified, ROUTE initially clears any previously existing routes for that device and then establishes the routes specified by dev2, dev3, etc., if any.

Any of the devices dev2, dev3, etc. may also be of the form MM=addr where addr specifies the main memory location of a user routine to which dev1 is to be routed. The first 12 bytes of the routine are reserved for use by DOS and must not be altered by the user. Upon routing, the user routine is entered via a CALL at the 13th byte, and it is the user's responsibility to save and restore all registers, except AF, used by the routine and routines it calls. If dev1 is an input device, the routine returns the new byte in register A with a zero indicating there is no new input byte from that routine. If dev1 is an output device, upon entry to the routine, register C contains the byte being outputted.

If dev1 is an output device, the output byte is sent to all routed-to devices in the order given in the ROUTE command.

If dev1 is an input device, each routed-from device is queried in the order given in the ROUTE command. If that device supplies a non-zero byte, the queries stop and the byte is used as the input byte for the dev1. If no routed-from device has an input byte, a zero is considered dev1's current byte.

The maximum number of routes-to and routes-from, excluding MM=addr types, in existence at one time is four for the Model I and six for the Model III.

WARNING!!! No editing of input or output characters is done during routing. This may cause problems (i.e., display control characters causing the printers to do unpredictable things).

ROUTE command examples:

1. ROUTE,PR,DO Printer output does not go to the printer but instead goes to the display.
2. ROUTE,DO,DO,PR Display output goes to both the display and the printer.
3. ROUTE,PR,DO,PR Printer output goes to both the display and the printer. If the routes of both example 2 and 3 are active, the routing is equivalent to the Model III TRSDOS function DUAL.
4. ROUTE,KB,RI (Model III only) Keyboard input characters come from the RS-232 input device and not from the keyboard.
5. ROUTE,DO,RO (Model III only) Display output is sent to the RS-232 output device and not to the display.
6. ROUTE,PR,MM=0FE80H Printer output is sent to the routine at main memory location 0FE80H (the routine's actual entry point is 0FE8CH).

7. ROUTE,KB,KB,MM=0F800H Keyboard input comes from either the keyboard or the routine at main memory location 0F800A. Input from the keyboard has precedence.

8. ROUTE,PR,NL Printer output is discarded.

9. ROUTE,PR All routing for the printer is dissolved.
Printer output goes to the printer.

10. ROUTE,CLEAR All routes are dissolved, and all devices are returned to their normal paths.

2.44. SETCOM (Model III only) Set RS-232 interface parameters.

SETCOM[,OFF][,WORD=w1][,BAUD=br][,STOP=sb][,PARITY=pp][,WAIT][,NOWAIT]

The SETCOM command optionally changes the state of the RS-232 interface and always displays the state. For RS-232 discussion, see chapter 8 of the Model III Operation and BASIC Language Reference Manual. The SETCOM command affects only the standard RS-232 control blocks and routines.

If OFF is specified, the RS-232 interface is turned off. No other optional parameters may be specified.

If any of WORD, BAUD, STOP or PARITY is not specified, the state for that keyword is not changed.

WORD=w1 specifies the number of bits per transmission byte. w1 must be one of 5, 6, 7 or 8.

BAUD=br specifies the transmission rate (the baud rate) for both sending and receiving. The 16 allowable values for br are 50, 75, 110, 134, 150, 300, 600, 1200, 1800, 2000, 2400, 3600, 4800, 7200, 9600 and 19200.

STOP=sb specifies the number of stop bits to be used for each byte transmitted. sb is either 1 or 2.

PARITY=pp specifies the parity to be used in the transmission where 1 = odd parity, 2 = even parity and 3 = no parity.

WAIT or **NOWAIT** are mutually exclusive and specify whether or not the RS-232 input routine is to wait until an input byte is received and the output routine is to wait until the current byte has been sent. If neither WAIT nor NOWAIT is specified, the previous wait or no wait state remains.

SETCOM command examples:

1. SETCOM,WORD=8,BAUD=30d,STOP=1,PARITY=1,WAIT Activates the RS-232 interface, if not already active, and sets the interface for 8 bit bytes. 300 baud rate, one stop bit, odd parity and forces the RS-232 routines, when called, to wait until an input byte is ready or until the RS-232 output device will accept an output byte.

2. SETCOM,NOWAIT,PARITY=3,WORD=7 Activates the RS-232 interface, if not already active, and sets the interface for 7 bit bytes, no parity and causes the RS-232 routines not to wait until an input byte is ready or the RS-232 output device will accept an output byte. The TRS-80 interrupt routines will handle the actual byte input or output with the RS-232 device. The other parameters not mentioned in the command are not changed.

3. SETCOM,OFF The RS-232 interface is deactivated. The current interface specification is remembered.

2.45. STMT Display specified message.

STMT,msg

Since normal DOS commands are always displayed, this command normally has nothing to do since its function, to display the message msg, has already been done. However, if this command was invoked via DOS-CALL (which does not display the DOS command), the message msg is displayed.

STMT is one of 3 ways in chaining to display a message without a pause. This allows multiple line instructions to be displayed, with the last line being a PAUSE and the others being STMTs.

STMT command examples:

1. STMT PHASE ONE COMPLETED This is simply an announcement to the terminal operator that phase one (whatever that was) has been completed. DOS does not pause.

2. STMT DISMOUNT AND STORE AWAY DISKETTE XXX
PAUSE AND MOUNT DISKETTE YYY ON DRIVE 2.

This example illustrates the combined use of the STMT and PAUSE commands to give instructions and wait until they are carried out.

2.46. SYSTEM Change system options.

SYSTEM,[password1:]dn1[,AA=yn][,AB=yn][,AC=yn][,AD=yn][,AE=yn]
[,AF=yn][,AG=yn][,AI=yn][,AJ=yn][,AL=al][,AM=am][,AN=an]
[,AO=ao][,AP=ap][,AQ=yn][,AR=yn][,AS=yn][,AT=yn][,AU=yn]
[,AV=av][,AW=aw][,AX=ax][,AY=yn][,AZ=yn][,BA=yn][,BB=yn]
[,BC=yn][,BD=yn][,BE=yn][,BF=yn][,BG=yn][,BH=yn][,BI=bi]
[,BJ=bj][,BK=yn][,BM=yn][,BN=yn]

The NEWDOS/80 system diskette whose control information is being updates/displayed by this command is mounted on drive dn1. If passwords are enabled, password1 must be specified and be equal to the diskette's master password. If no optional parameters are specified, then only a display of existing options is given. The optional parameters may be specified in any

order, and only those parameters specified have their values changed in the diskette's control data (3rd sector on the diskette). Parameters not specified are not changed.

If many options are being changed, it may be necessary to perform multiple SYSTEM commands as the DOS buffer is limited to 79 characters per command.

It is anticipated that additional options will be specified as time proceeds.

Changes to a system diskette's system options do not affect the computer operations until that system diskette is mounted on drive 0 and a reset done.

AA=yn If AA=Y, passwords are enabled. If AA=N, passwords are disabled.

AB=yn If AB=Y, the system is to operate in RUN-ONLY mode. SYSTEM options AD=N, AE=N and AF=N are forced at reset time, and the pressing of ENTER to override the auto command is disallowed. The user must have a proper auto command (see AUTO, section 2.4) that will either invoke a user program or execute a CHAIN file that will eventually invoke a user program. In RUN-ONLY mode, if the system finds itself at normal DOS READY or MINI-DOS READY, it will go into an endless loop after displaying 'RUN ONLY STOPPEDI! PRESS 'R' FOR RESET'. Upon receiving R, the DOS command BOOT (see section 2.7) will be executed. BASIC honors RUN-ONLY by disabling BREAK, treating LOAD without R or V as an error, and by not allowing any direct statements. If AB=N, the system is in normal command mode.

AC=yn (Model I only) If AC=Y and if SYSTEM option AJ=Y, the NEWDOS/80's debounce routine is used. If AC=N or SYSTEM option AJ=N, the NEWDOS/80's debounce routine is bypassed.

AD=yn If AD=Y, 'JKL' is enabled, and if AD=N, 'JKL' is disabled.

AE=yn If AE=Y, '123' is enabled as the method to invoke DEBUG (see section 4.1). If AE=N, '123' is disabled.

AF=yn If AF=Y, 'DFG' is enabled as the method of invoking MINI-DOS (see section 4.2). If AF=N, 'DFG' is disabled.

AG=yn If AG=Y, BREAK is considered a normal input key with code = 01. If AG=N, the BREAK key is not considered a normal input key and its occurrence is changed to the null key code 00. The state of the BREAK key is set according to option AG at reset and then again every time the system returns to normal DOS READY. DOS command BREAK may be used to enable or disable the BREAK key until the next normal DOS READY. Also, programs may enable the BREAK key by storing a 0C9H byte in Model I location 4312H (Model III location 4478H) or disable the BREAK key by storing a 0C3H byte in that location.

AH=yn Not defined in NEWDOS/80, version 2. Formerly, this dealt with delaying the disabling of timer interrupts during disk I/O to gain better clock accuracy. This is no longer done.

AI=yn (Model I only) If AI=Y, lower case modification has been installed in the computer and AI=N it is not. User programs may test for bit 4 of 436CH for this state, 1 if AI=Y and 0 if AI=N. Currently, DEBUG and SUPERZAP use this flag to decide whether memory displays can display lower case.

AJ=yn If AJ=Y, NEWDOS/80's keyboard intercept routine is active. This routine contains repeat key function, 'debounce' (Model I only) and one of the methods used to spot 'JKL', '123' and 'DFG' (the other being off the timer interrupts). If AJ=N, NEWDOS/80 does not intercept the keyboard two byte address vector at 4016H and

1. The repeat key function for the Model I is not active regardless of the SYSTEM option AU. The Model III reverts to the ROM repeat key function.
2. 'debounce' (Model I only) is not active regardless of SYSTEM option AC setting.
3. 'JKL', '123' and 'DFG' can only be triggered via the interrupts, resulting in many more spurious key input characters.

If the up-arrow key is depressed all during the reset/power-on sequence, AJ=N is forced; this is necessary for those programs that eventually overlay the DOS in main memory.

AK=yn Not defined in NEWDOS/80, version 2. Formerly, this option dealt with allowing 'JKL' to pass graphic characters to the printer. This has been incorporated into SYSTEM option AX.

AL=al al (value 1 - 4) specifies the number of physical drives in the system. If your system only has one drive, setting al = 1 will limit the system to only checking for that one drive. Though al can be set to 255, it should never exceed 4.

AM=an an (value 0 - 255 where 0 = 256) is the number of tries allowed for a disk I/O before it is declared in error. The original DOSS used a value of 10.

AN=an an = the default drive number for the DIR command.

AO=ao When creating a file and when the user lets the system choose the diskette to contain the file by not specifying a drive number in the filespec, the system will first search all the drives for an existing copy of the file. If it does not find an existing copy, the system will start searching at drive so, and will search that and higher numbered drives until a free FDE is found. It will not search a drive whose number is less than ao.

AP=ap ap is a memory address, which if other than 0 and is within the range of existing memory, is stored as DOS's HIMEM address value in the two bytes at Model I location 4049H (Model III location 4411H).

AQ=yn If AQ=Y, the CLEAR key is enabled, and if AQ=N, the CLEAR is disabled if SYSTEM option AJ=Y.

AR=yn If AR=Y, COPY, formats 5 and 6, are allowed without diskette password checking even though passwords are enabled. If AR=N, passwords are required if passwords enabled.

AS=yn (Model I Only) If AS=Y, BASIC will convert input text character strings from lower to upper case. This is useful when lower case hardware is not installed or when lower case drivers are not used as it is very possible to input lower case characters (using the shift key) and have BASIC display them as upper case even though they are really lower case. The user can stare forever at a compare that looks equal on the display, but BASIC computes as unequal. If AS = N, BASIC will leave the text character strings alone. This option does not affect string characters input as data rather than as part of text.

AT=yn AT=N puts chaining into record mode, meaning that only requests for full records come from the chain file; single char key input request are honored from the keyboard. AT=Y puts chaining in single character mode meaning that all requests for an input key come from the chain file.

AU=yn AU=Y turns on the clock driven repeat key function. The first repeat will delay option AV number of 25 ms intervals. Subsequent repeats will enter as fast as the program asks for them but not more than 12 per second. AU=N turns off the repeat key function, eliminating repeat keys on the Model I and shifting to the ROM repeat key function on the Model III.

AV=av AV is used when AU=Y. av is the number of 25 ms intervals to pass between the key depression and the acceptance of the 1st repeat of that character. Subsequent repeats are as fast as the program wants them but not more than 12 per second.

AW=av is the number of write-with-verify disk I/O tries allowed. This I/O retry count works in conjunction with option AM=am with each retry under AW taking place only after the sector verify read has failed am number of times. Formerly, if sector write encountered no error and the verify read did result in an error, it was left to the user to retry the write. Now, if aw is greater than 1, the write will automatically be retried in the cases where the write was apparently good but the verify read failed.

AX=ax This is ASCII code of the highest printable character for the printer. It is used by system routines to determine when to substitute blanks or periods in place of ASCII codes higher than this value. This value must not exceed 255. This high ASCII code is stored in the one byte at Model I location 4370H (Model III location 4290H).

AY=yn is used only during resets wherein DOS senses that it was not active immediately prior to the reset (i.e., reset after power-on or after execution of non-disk BASIC). AY=Y causes the operator to be asked for date and time. AY=N bypasses this query and causes date and time to be set to zeroes.

AZ=yn is used only during resets wherein DOS senses that it was active immediately prior to the reset. AZ=Y causes the operator to be asked for date and time. AZ=N causes date and time to be left as they were prior to the reset.

BA=yn BA=Y causes a reset to activate 'ROUTE,DO,NL', thus causing all display output, including the DOS and BASIC banners, to be lost until the operator or a user program executes either 'ROUTE,CLEAR' or 'RDUTE,DO'. BA=N disables this reset action.

BB=yn (Model III only) BB=N informs the system that the clock interrupts

occur 60 times a second. BB=Y informs the system that the clock interrupts occur 50 times a second. This option does not set the clock to perform as such, but only acknowledges that it does.

BC=yn BC=Y means the operator can manually pause or cancel chaining. BC=N means the operator is not allowed to manually pause or cancel chaining. RUN ONLY forces BC=N.

BD=yn BD=Y means the operator can override the AUTO command at reset by holding down the ENTER key. BD=N means he/she can't. RUN ONLY forces BD=N.

BE=yn BE=Y enables the DOS command R to repeat the previous DOS command (see section 2.41). BE=N causes the R command to simply return to DOS READY.

BF=yn (Model I only) BF=Y performs at reset/power-on time the equivalent of the DOS command LCDVR,Y (see section 2.29). BF=N performs the equivalent of LCDVR,N. However, if DOS senses that the lower case hardware is either not installed or is not operating, BF=N is forced.

BG=yn BG=Y performs at reset/power-on time the equivalent of the DOS command LC,Y (see section 2.28). BG=N performs the equivalent of LC,N

BH=yn At reset/power-on time BH=Y enables cursor blinking, and BH=N inhibits it.

BI=bi At reset/power-on time, the numeric value bi is set as the cursor character's value, excepting that if bi = 0, then the standard cursor character value is used (95 for the Model I and 176 for the Model III).

BJ=bj Option BJ provides a minimal control for NEWDOS/80 when a CPU speed up modification is installed that is to continue operation during disk operations. This option multiplies (roughly) by bj the number of Z-80 instructions executed during certain timing loops used internal to NEWDOS/80. bj must be an integer greater than 0 and equals the number of times the CPU has been speeded up. Set bj = 1 if the loops are not to be lengthened. If the loops are to be lengthened, bj must always be rounded up in the cases where the new CPU speed is not an even multiple of the original Model I or Model III speed. Option BJ does NOT perform the actual CPU speed switching.

BK=yn BK=Y allows the DOS command WRDIRP and the W and C functions of DIRCHECK to be executed. BK=N causes these functions to be rejected with 'DISK ACCESS DENIED'.

BM=yn BM=Y causes diskette formatting to verify read sectors in a separate VERIFYING phase after all tracks have been formatted. This verify read is in addition to the verify read done on a track's sectors immediately after the individual track was formatted. BM=N bypasses this VERIFYING phase, deeming as sufficient the verify sector read done when the individual track was formatted.

BN=yn (Model I only) BN=N causes the write of single density diskette directory sectors to use the address mark readable by Model I TRSDOS. BN=Y causes the write of single density diskette sectors to use the address mark readable by Model III NEWDOS/80. BN=Y should only be used where it is required that single density diskettes be NEWDOS/80 version 2 exchangeable between the Model I and the Model III.

Though the information contained in the directories used by Model I TRSDOS, Model I NEWDOS/80 and Model III NEWDOS/80 is the same (except for some additions by NEWDOS/80), the address mark byte (part of the magnetic format and identification bytes that surround each 256 bytes of user data on the soft sector diskettes) used to indicate the directory sectors are 'protected' is different on the Model III than it is on the Model I for single density diskettes.

The changing of SYSTEM option BN does not in itself change the address mark of any directory sectors. All this does is set the protected sector write routine in DOS to write the specified address mark Whenever a protected sector is written or rewritten to disk. To set all sectors of a single density diskette directory to the proper address mark, use either DOS command WRDIRP or DIRCHECK with the W option. **Warning!!!** If a single density diskette has been used on the Model III or has been used on the Model I where BN=Y and the diskette must now be used with Model I TRSDOS, the user must set BN=N and rewrite the directory sector address marks using WRDIRP or DIRCHECK with option W. This must be done even though, with BN=N, SUPERZAP under NEWDOS/80 on the Model I shows the directory sectors protected; this is because Model I NEWDOS/80 accepts either address mark value as 'protected' though it only writes the one value specified by option BN.

System option codes B0 and up are reserved for future definition.

SYSTEM command examples:

1. SYSTEM,0,AL=4,AA=Y,AU=Y,AV=20,AT=Y The SYSTEM control parameters AL, AA, AU, AV and AT are changed on the current system diskette mounted on drive 0. All the other SYSTEM parameters are left unchanged. The full SYSTEM specification is then displayed. These changes are not used to control NEWDOS/80 until the next reset/power-on.

2. SYSTEM,2,AP=0FF0AH,AN=1,AX=126 The SYSTEM control parameters AP, AN and AX are changed in the control sector of the diskette currently mounted on drive 2. No other SYSTEM parameters are changed. The full system specification contained on that diskette is then displayed. For the SYSTEM parameters contained on that diskette to control NEWDOS/80, that diskette must be a NEWDOS/80 version 2 system diskette, must be dismounted from drive 2 and remounted on drive 0, and a reset/power-on must be done.

2.47. TIME Set the real time clock.

TIME[,hh:mm:ss]

If no parameters are specified, the current times is displayed in hh:mm:ss format.

If hh:mm:ss is specified, the clock is set to time hh:mm:ss where hh is a 2 digit hour value, 00 - 23, mm is a two digit minute and ss is a two digit seconds value. No check is made op the validity of the values. Each of the three values is converted to a single byte value and stored into its byte of

the clock. The clock three bytes start at model I location 4041H (model III location 4217H) and are in seconds, minutes, hours order.

At reset/power-on the clock is set according to SYSTEM option AY or AZ. The clock is updated once a second. The user should not rely upon the clock for an accurate value as disk I/O frequently and interrupt routines infrequently run so long with interrupts disabled that one or more timer interrupts will be missed, causing the clock to run slow. The real time clock is not a hardware clock, but instead is maintained by software that is not aware of the lost timer interrupts.

TIME command examples:

1. TIME,15:23:00 The clock is set to 3:23 PM.
2. TIME The current time is displayed.

2.48. VERIFY Require verify read after every disk write.

VERIFY[,yn]

NEWDOS/80 performs verify read after all of its directory writes and after all sector writes when logical record or single byte I/O is used. It does not perform verify reads when full sector writes are done via the 4439H vector.

VERIFY or VERIFY,Y Diskette writes done via the 4439H vector are verify read. A verify read means the sector is read after it is written. If the sector was written illegible or with bad parity, an error will be triggered. A byte for byte data compare is not done. However, if the verify read detects an error and SYSTEM option AW is not equal to 1, the write and verify read will be done again since the system still has access to the data that should have been placed into the diskette sector.

VERIFY,N Diskette full sector writes done via the 4439H vector are not verify read.

COPY, EDTASM and BASIC SAVE's write the file completely without validity read, but then read back the entire file as a verify read. All BASIC disk data writes to print/input files, marked item files, fixed item files or field item files (where record length is not 256) perform verify read due to the fact that byte rather than sector I/O is used. Field item files with record length 256 use sector I/O and are not verify read unless VERIFY is on.

2.49. WRDIRP Write directory sectors protected.

WRDIRP,dn1

WRDIRP causes the directory sectors for the diskette in drive dn1 to be read and rewritten in the currently defined protected state for the current computer (see SYSTEM options BN and BK).

This command is used where single density diskettes are to be exchanged under NEWDOS/80 version 2 between the model I and III.

This command enables the user to set the directory to the proper read protect state while under MINI-DOS, since it is most likely he/she will find out about the problem when in the middle of doing something else (and thus can't get to DIRCHECK). **CAUTION!!!** This command uses the directory starting granule number from the 3rd byte of the boot sector to find the directory. It then checks to see if the FPDE's for BOOT/SYS and DIR/SYS are present. If these checks pass, it then changes what it thinks are the directory sectors all to protected status. Do NOT use this command unless you are sure the only problem is the different protection status between the model I and model III; if you have doubts, use the W function of DIRCHECK.

If SYSTEM option BK = N, the DOS command WRDIRP is disabled.

WRDIRP command example:

WRDIRP,1 For the diskette mounted on drive 1, the directory address marks are set for the current computer and, if Model I, for the setting specified by SYSTEM option BN.

3. DOS ROUTINES

3.1. Specifications Defined

This chapter specifies the DOS routines that are available for use by machine language programs. If you are neither a Z-80 programmer nor interested in Z-80 machine code, you should bypass this chapter. Readers of this chapter are assumed to be knowledgeable of Z-80 machine code and at least one assembly language for the Z-80.

These DOS routines have entry and exit conditions, and rather than repeat them in each routine's specification, some of the conditions are defined here with the using routine's specification simply referring to the condition's code.

- A. Only register AF is altered by the routine. Any other registers used by the routine are saved on entry and restored on exit.
- B. On exit, Z state is set if no error is encountered during the routine's execution. NZ state is set if a DOS error is encountered, and register A contains a DOS error code. The setting of Z and NZ takes precedence over the setting of other flags such as C and NC.
- C. On entry, DE points to an open FCB.

There are incompatibilities with TRSDOS in the use of some of these routines. They are discussed briefly in the routines where they occur, so study them carefully. The reader should also be aware of the differences in the way the FCB fields NEXT and EOF are maintained (see FCB specification, section 5.9).

The discussion of each routine gives its entry address (the address to be used in the CALL or JP Z-80 instruction), then its title (if one is appropriate), and then its specification.

Unless otherwise specified, the DOS routine uses the invoker's stack. Unless specified as a dead end routine, the DOS routine exits to the caller.

Many of these routines use a FCB (see section 5.9). NEWDOS/80 on both the Models I and III and Model I TRSDOS all use a 32 byte FCB while Model III TRSDOS uses a 50 byte FCB. NEWDOS/80 will run with user programs having the 50 bytes FCB but will only use the first 32 bytes of those FCBs. Programs using a 32 byte FCB with Model III TRSDOS will have problems.

The routines listed below are not necessarily in ascending numeric order.

3.2. 402DH No-Error Exit

Dead end routine. Programs concluding with no error jump to 402DH. DOS checks its own state in the following order.

If either MINI-DOS or DOS-CALL, the stack pointer is set to where it was before the last DOS command; otherwise it is set to DOS's stack area and the BREAK key is enabled/disabled according to system option AG.

If DOS-CALL and if either not chaining or chaining is not to be continued at the current DOS level, all registers except AF are restored to as they existed on DOS-CALL entry, Z state is set, and a return is made to the DOS-CALL invoker. If this was the outermost DOS-CALL level, DOS is taken out of DOS-CALL state.

If RUN-ONLY and if chaining is not active, the message 'RUN ONLY STOPPED!! KEY 'R' FOR RESET.' is displayed, DOS loops waiting on the reply, and then executes DOS command BOOT (see section 2.7).

If DOS-CALL and if chaining is to continue at the current DOS-CALL level, DOS waits for the next command from the chain file.

If MINI-DOS, then MINI-NEWDOS/80 READY is displayed, and DOS waits for the next command.

If chaining is active, DOS waits for the next command from the chain file.

NEWDOS/80 READY is displayed and DOS waits for the next input command.

3.3. 4030H Error-already-displayed DOS Error Exit

Dead end routine. Programs concluding with an error that is either already displayed or not to be displayed jump to 4030H. DOS action is the same as for 402DH except as follows:

If CHAINING, chaining is aborted.

If DOS-CALL, the current DOS-CALL level is exited in the same manner as for 402DH, except that C state is set.

3.4. 4400H No-Error Exit. Performs identical to 402DH.

3.5. 4405H Enter DOS and execute a command

Dead-end routine. DOS is entered, and the stack pointer is set to DOS's own area. HL points to a command, terminated by a 0DH byte, that DOS is to use as its next command. DOS moves this command to its own 8\$ byte command buffer and then executes it.

3.6. 4409H DOS Error Exit

Dead end routine if bit 7 of register A equals 0. Programs terminating with a DOS error jump to 4409H with the DOS error code in register A and bit 7 of register A equal 0. Depending upon DOS's state, the following actions occur:

If CHAINING, chaining is aborted.

If DOS-CALL, the current DOS-CALL level is exited in the same manner as for 402DH exit, except NZ and NC state is set and the DOS error code is in register A. The error msg is not displayed.

Otherwise the DOS error message is displayed, and an exit is taken to 402DH.

A program may CALL 4409H to display an error msg by placing the error code in A and setting bit 7 of register A equal to 1. The appropriate DOS error message will be displayed. On return, only the F register has been altered.

The Model I TRSDOS will print diagnostics if bit 6 of register A equals 0. The Model III TRSDOS displays only the error number if that bit equals 0 and the error message if that bit equals 1. NEWDOS/80 ignores the value of that bit.

Debugging hint. By setting the 4 bytes at 4409H equal to CD 0D 44 C9, the error display routine can be made to invoke DEBUG instead of displaying the error message.

3.7. 440DH Enter DEBUG

User programs have two methods of entering the DEBUG facility: (1) by use of Z-80 instruction RST 30H and (2) by the Z-80 instruction CALL 440DH. When done with the DEBUG facility, DEBUG command G will return to the instruction following the RST 30H or the CALL, provided the PC register was not changed.

3.8. 4410H (447BH in Model III) Enqueue a user timer interrupt routine.

Registers AF, BC, DE and HL are altered by this routine. On entry, DE points to the user interrupt routine, which must conform to the following format:

1st 2 bytes. Used by DOS as a forward chain pointer. On entry, the two bytes can be any value.

3rd byte. The number of 25ms intervals to pass between invocations of the user's routine. Example, if the routine is to be invoked every second, the 3rd byte must be set = 40 (28H). DOS does not alter this byte.

4th byte. Count down value to the next invocation. On entry, this byte should be properly initialized to a value greater than 0 but less than or equal to the value in the 3rd byte. Every 25ms interrupt, DOS decrements this value. If the result is non-zero, this routine is bypassed for this 25ms interrupt. If the result = 0, the value from the 3rd byte is moved into the fourth byte, registers HL, DE, BC and AF are saved, and the user routine is called at its 5th byte. Any other registers used by the routine must be saved/restored by it. Interrupts are disabled, and the user routine must not re-enable them.

While a user interrupt routine is in the interrupt chain, it must not be altered in any way except by a routine that runs with interrupts disabled; the first two bytes must never be altered.

Model I TRSDOS uses the 4 vectors, 4410H, 4413H, 4416H and 4419H, for its user interrupt routine handling. NEWDOS/80 uses only 4410H and 4413H for non-compatible handling of these routines. Any program using a 25ms interrupt user routine in TRSDOS must be modified to work under NEWDOS/80. This is a major incompatibility between the two Model I systems.

Model III TRSDOS has not yet made any provision for user timer routines, using 4410H - 441BH for other purposes, including HIMEM.

Model III NEWDOS/80 continues with the user timer interrupt routine mechanism used on the the Model I, except that 447BH is the routine enqueue vector instead of 4410H, and in order to continue with 25 ms counting where the Model III clock actually counts in either 30ths or 25ths of a second, a second pass through the user routine check and invocation sequence is done when necessary to bring 25ms counting up with the real clock. If a user routine is being invoked every 25 ms, the routine must be prepared to accept two invocations within the same timer interrupt.

3.9. 4413H Dequeue a user timer interrupt routine.

Registers AF, BC, DE and HL are altered. The user interrupt routine (as described in section 3.7) pointed to by register DE is taken out of the 25ms interrupt chain, if it is in the chain. The routine no longer participates in the interrupts and may now be altered at will by the user.

See section 3.8 for TRSDOS incompatibility.

3.10. 4416H Keep drives rotating

If the disk drives are rotating, reselect the current drive, thereby keeping the drives rotating for approximately 2.4 seconds more. Register AF is altered.

This routine does not exist in TRSDOS; see section 3.8 for incompatibility.

3.11. 4419H DOS-CALL Execute a DOS command and return.

This routine is DOS-CALL. DOS does not shift to its own stack area, but instead remains with the user's stack. All registers except AF are saved in the stack and will be restored on return. The command to be executed is pointed to by HL, must be less than 80 characters, must terminate with byte 0DH, and can be anything legal for the current state DOS is in. DOS sets DOS-CALL state, if not already set, saves the current stack pointer, and executes the command. The command can be the invocation of a user program.

DOS-CALL is now legal under CHAINING where it was not in NEWDOS/80 Version 1.

DOS-CALL is the way BASIC executes the DOS command contained within the BASIC statement `CMD"xx"` where `xx` is the DOS command.

The DOS-CALL caller is responsible for assuring that memory conflicts do not arise and that sufficient stack space is available.

Nested calls to DOS-CALL may be executed. Upon exiting from a DOS-CALL level, the return is made to the next outer level. When the outermost level is exited, DOS leaves DOS-CALL state.

If the DOS command invokes a program, that program may use its own stack area, and it must exit using one of the three exits: 402DH, 4030H or 4409H. On exiting, the program may store a 2 byte parameter in 4403H, 4404H (17411, 17412 decimal) for use by the caller.

The 4419H vector is used differently in TRSDOS; see section 3.8 for incompatibility.

See section 4.4 for further discussion of DOS-CALL.

3.12. 441CH Extract a filespec

From the text pointed to by HL, extract a filespec, place it in the area pointed to by DE and terminate it with the byte 03H. Registers AF, BC and HL are altered.

If the first text character is A - Z or 0 - 9, or if the first text character is * and the next character is A - Z or 0 - 9, text is moved from the HL area to the DE area until a character that is not /, ., :, A - Z, or 0 - 9 is encountered or until 32 bytes have been transferred. If less than 32 bytes, a 03H byte is placed after the last byte in the DE area to indicate end of filespec, and a return is made with Z state set. If the filespec is more than 31 characters it is considered improper as discussed in the following paragraph.

If the first character was improper, or if the first character was * but the 2nd was improper, a return is made with NZ state set.

On exit, if the terminator/improper byte equals 03 or 0DH, then HL points to that byte; otherwise HL points to the next byte.

The user will notice that NEWDOS/80 doesn't check for an exact filespec; it leaves this to be done by the OPEN routines, 4420H and 4424H.

3.13. 4420H Open a FCB to a new or existing disk file

Conditions 3.1.A and B hold. The entry requirements are the same as for 4424H, which is executed immediately as a subroutine to this routine. If 4424H is successful in opening an existing file, no further action is required here, and an exit is taken with Z and NC states set. If the file was not found, this routine proceeds to create the file.

If the filespec in the FCB pointed to by register DE specifies an explicit

drive number and the diskette mounted on that drive has a free FDE, the file is created on that diskette whether or not the diskette actually has any free space. If the filespec did not specify a drive number, the system starts searching mounted diskettes, starting with the drive number specified by SYSTEM option AO and preceding through higher numbered drives until a diskette with a free FDE is found. If a free FDE is not available, the file cannot be created, and the error exit is taken.

Creating a file consists of converting a free FDE to a FPDE. This entails inserting the name and name extension (if any), encoding the password (if any) as both the update and access passwords, storing the LRECL (0 means 256) from register B, setting the EOF equal to 0, setting access level as FULL, and marking the file non-system, non-invisible. No diskette file space is assigned to the file at this time; in fact, DOS doesn't even look to see if the diskette has any free space. Note, though the LRECL is stored in the FPDE during file creation, it is never used. Each subsequent open of the file uses the LRECL provided in register B.

After the file is created, the DOS routine at 4424H is called to perform the OPEN. On exit after a successful file create and open, Z and C states are set.

3.14. 4424H OPEN a FCB to an existing file

Conditions 3.1.A and B hold. On entry, register DE points to a FCB containing the filespec for the file to be opened, HL points to a 256 byte buffer to be used during disk sector reads and writes for this FCB, and B contains the LRECL (0 = 256). If an explicit drive number was specified in the filespec, the search for the file is limited to that drive; otherwise the search starts with drive 0 and proceeds to higher drives until a file with the specified name and name extension is found. If no file is found, the error exit is taken.

If passwords are enabled and the file has non-null passwords, then an error exit is taken if the filespec does not contain either the update or the access password. If passwords are disabled or the file has no passwords or the update password is specified, the FCB's access level is set to FULL; otherwise the access level from the FPDE is placed into the FCB to limit the type of access for this file.

The FCB is converted from containing the filespec to containing information about the file, which will be used while the FCB is open to reduce the amount of directory I/O which would otherwise be required. The conversion entails copying the EOF and the 1st 4 extents from the FPDE, storing the LRECL from register B, setting bit 7 of the FCB's 2nd byte equal to 1 if LRECL is not equal to 0 (to indicate logical record processing), setting NEXT equal to 0, storing the drive number and the FPDE's DEC code, storing the 256 byte buffer pointer from register HL, setting the access level, setting bit 5 of the FCB's 2nd byte equal to 1 to indicate that the buffer does not contain the current, sector and setting bit 7 of the FCB's 1st byte equal to 1 to indicate that the FCB is open.

3.15. 4428H CLOSE a FCB. Conditions 3.1.A, B and C hold

This routine dissolves the connection between the FCB and the file. If bit 4 of the FCB's 2nd byte equals 1, the FCB's buffer is written to disk like a 4439H call. If the FCB's EOF is different from that in the FPDE, the FPDE is updated for the new EOF. If the file has excess granules beyond EOF and if automatic space deallocation is allowed, the excess granules are released. The FCB is then converted back to contain a filespec consisting of the file name, name extension (if non-blank) and the drive number. This filespec can be used later to re-open the file, provided a password is not required.

3.16. 442CH Kill the FCB's associated file

Conditions 3.1.A, B and C hold. The file associated with the FCB is killed in the same manner as for DOS library command KILL (see section 2.27). The FCB is set to all zeroes.

3.17. 4430H Load a program file

Conditions 3.1.A and B hold except the registers AF, BC and HL are altered and on exit HL (and 4403H - 4404H (17411 -17412 decimal)) contain the program's entry address. On entry, register DE points to a FCB containing the program file's filespec. The load is done the same as for DOS library command LOAD (see section 2.32).

3.18. 4433H Load and commence execution of a program file

Dead end routine. On entry, DE points to a FCB containing the program file's filespec. Registers AF and BC are altered; all other registers are passed on unchanged to the program when its execution begins. The file open, load and commence execution are done the same as when DOS executes a command that is not a library command, excepting that there is no default name extension. If an error occurs during the open or load, DOS exits to 4409H. If DEBUG is active (see section 2.17), DEBUG is entered just before the program commences execution.

3.19. 4436H Read sector or logical record from disk

READ a disk sector or move a logical record from the FCB's buffer to the caller's buffer. Conditions 3.1.A, B and C hold. If bit 7 of the FCB's 2nd byte equals 0, the sector represented by the high two bytes of the NEXT field is read into the FCB's buffer and, if no error or if error code 6 (sector read protected), the NEXT field is advanced 256 bytes. If an error other than code 6 occurs, the NEXT field is not advanced, meaning the user can retry to read the same sector.

If bit 7 of the FCB's 2nd byte equals 1, then a logical record of length equal to the FCB's LRECL (where 0 means 256) is moved from the FCB's buffer to the buffer pointed to by register HL on entry. As each byte is moved, the NEXT field is incremented. When the FCB's buffer is empty, the next file sector is automatically read into it and byte movement continues. If an error

occurs, including error code 6, the logical record move terminates, leaving NEXT advanced for the number of bytes moved.

If bit 1 of the FCB's 1st byte equals 1, the NEXT and EOF fields are considered RBA's within the diskette rather than within a file, thus giving the user the capability to read a diskette, rather than a file. The use of bit 0 of the FCB's first byte is defined in section 3.20 below. DOS routines 0013H, 001BH, 4439H, 443CH and other routines that indirectly read or write sectors also operate as such if any of these two bits are on. The use of these 2 bits is incompatible with TRSDOS.

One incompatibility between NEWDOS and TRSDOS occurs when the program reads the EOF from the FCB to determine the number of bytes in the file. However, in many cases the user does not have to know what the EOF is. Instead, for both TRSDOS and NEWDOS, the user can read the file sector by sector, waiting for either of the two EOF errors. If the error code is 1CH (END OF FILE ENCOUNTERED), then the file ends on a sector boundary and the last sector read successfully was the file's last. If the error code was 1DH (PAST END OF FILE), then the last sector successfully read was also the file's last, but was only a partial sector with the value in FCB+8 equaling the number of bytes in that sector belonging to the file. Remember, this is true for both TRSDOS and NEWDOS; thus the same code can work for both.

3.20. 4439H Write sector or logical record to disk

WRITE without verify a sector to disk or move a logical record from the caller's buffer to the FCB's buffer. Conditions 3.1.A, B and C hold. IF bit 7 of the FCB's 2nd byte equals 0, the disk sector as defined by the NEXT field is written with the contents of the FCB's buffer. Unless VERIFY is on (see section 2.48), verify read is not done. If no error, and if the lower order byte of NEXT equals 0, the NEXT field is advanced 256 bytes. Whether or not NEXT was advanced, if NEXT now exceeds EOF or if bit 6 of the FCB's 2nd byte equals 0, EOF is set equal to NEXT. If an error occurred, NEXT is not altered, thus allowing the user to retry to write the same sector.

If bit 7 of the FCB's 2nd byte equals 1, a logical record of length equal to the FCB's LRECL (0 means 256) is moved from the caller's buffer, pointed to by register HL on entry, to the FCB's buffer. With each byte's move, NEXT is incremented, and if NEXT now exceeds EOF or if bit 6 of the FCB's 2nd byte equals

EOF is set equal to NEXT. When the FCB's buffer fills, the buffer is written to the appropriate disk sector with verify read and then the logical record move continues, filling in the FCB's buffer for the next file sector. Whenever an error occurs, the logical record move terminates, leaving NEXT advanced for the number of bytes moved.

Bit 1 of the FCB's 1st byte functions as described in section 3.19. If bit 0 of that byte equals 1, then sectors are written protected (error code 6 on sector read).

If a verify read is done after the write of a protected sector, error code 6 is not returned to the caller as an error.

A significant incompatibility with TRSDOS lies in the fact that when a sector

is written to disk in NEWDOS/80 and the low order byte of NEXT is non-zero, NEXT is not advanced by 256 bytes. In this case, NEWDOS/80 assumes that the caller is writing the last sector of the file (though it need not be) that is only partially full, and that NEXT already is the proper RBA value for EOF (if EOF is to be updated by the write).

One incompatibility between NEWDOS and TRSDOS is in setting the final EOF for a file that is written sector by sector but usually does not end on a sector boundary. However, if the program knows when it is about to write the last sector, whether partial or full, and can store the desired low EOF byte value in FCB+5 just before writing that last sector, both TRSDOS and NEWDOS will exit from that write with the same EOF. Thus, in this instance, the same program code will work for both TRSDOS and NEWDOS, and no incompatibility exists.

3.21. 443CH Write sector or logical record to disk with verify read

This routine is identical to 4439H, except that a verify read is always done after a sector write.

3.22. 443FH Position FCB to start of file

Conditions 3.1.A, B and C hold. If the FCB has a sector awaiting write (bit 4 of FCB 2nd byte = 1), it is written as a 4439H call. The FCB NEXT field is set = 0, Bit 5 of FCB 2nd byte is set = 0 to indicate the buffer does not contain the current sector.

3.23. 4442H Position FCB to a specified file record

Conditions 3.1.A, B and C hold. The NEXT field is set to the RBA of the logical record whose relative record number U = the first record) is in register BC upon entry. If the new NEXT is in the same sector as the old NEXT, the status of the current sector is not changed (i.e., the sector is not written to disk if bit 4 of the FCB 2nd byte equals 1). If the new NEXT is not in the same sector as the old NEXT, then (1) if bit 4 of the FCB 2nd byte equals 1, the old sector is written back to disk, and (2) bit 5 of the FCB 2nd byte is set to 1 to indicate that new sector has not yet been read into the buffer.

3.24. 4445H Position FCB back one record

Conditions and performance are the same as 4442H except that the NEXT field is reduced by the LRECL.

3.25. 4448H Position FCB to EOF

Conditions and performance are the same as 4442H except that the NEXT field is set equal to the EOF field.

3.26. 444BH Allocate file space

Conditions 3.1.A, B and C hold. If the file sector represented by the two high order bytes of the FCB's NEXT field is not already allocated to the file, the granule containing it is allocated along with the granules for any lower sectors for the file that are not yet allocated. This allows the programmer to allocate file space before it is actually needed, and is especially valuable when it is necessary to know that a sector can be written before any data is placed in the buffer. If a file's size can be predetermined before being written (such as is done in COPY), pre-allocating the necessary granules saves considerable time over allocating the granules as the file write proceeds.

This address is defined differently in TRSDOS.

3.27. 444EH Position FCB to the specified RBA

Conditions and performance are the same as for the 4442H call except the new NEXT position value is taken from the registers H, L and C where H contains the high order and C the low order values.

This address is defined differently in TRSDOS.

3.28. 4451H Write the EOF value from the FCB to the directory

Conditions 3.1.A, B and C hold. If the EOF value in the FCB differs from that in the file's FPDE, the FCB's EOF value is written into the FPDE on disk.

This address is defined differently in TRSDOS.

3.29. 445BH Select and power up the specified drive

Conditions 3.1.A and B hold. On entry, register A contains a drive number. That drive becomes the current drive, is selected and, if necessary, powered up.

3.30. 445EH Test for mounted diskette

Conditions and performances is the same as for 445BH excepting that, in addition, the drive is tested to determine if a diskette is mounted and is rotating. If this rotation test fails, error code 08, DEVICE NOT AVAILABLE, is returned.

3.31. 4461H *Name routine enqueue

Register HL points to a user routine in main memory to be chained in the chain of user logical routines. The first 12 bytes of the routine are defined as follows:

4 bytes reserved for use by DOS only.

8 byte logical routine name field containing the 1 - 8 character name of the routine, padded on the right with blanks.

If a routine with the same name already exists in the queue, FILE ALREADY EXISTS error code is returned with NZ set. Otherwise, the routine is enqueued, and exit taken with Z state set. HL, DE, BC and AF are altered by this function. This function is new with NEWDOS/80.

Subsequently, whenever a DOS command of the form *name1 or *name1,parameters is executed, DOS searches its queue for a routine named name1, sets HL pointing to the parameters, if any, and jumps to the routine's 13th byte. When the routine concludes, it should exit via 402DH, 4409H, or 4030H. The routine may use all registers, and can use the two bytes at 4403H - 4404H to receive or pass back a parameter. If the logical routine name1 does not exist in the queue, FILE NOT IN DIRECTORY error code is returned with NZ set.

3.32. 4464H *name routine dequeue

HL points to a logical routine as defined in section 3.31. If the routine is not in DOS's logical routine queue, this function exits with FILE NOT IN DIRECTORY error code in register A and with NZ set. Otherwise, the routine is dequeued, meaning that subsequent *name1 commands naming it will abort, displaying FILE NOT IN DIRECTORY. Registers HL, DE, BC and AF are altered by this function. This function is new with NEWDOS/80.

3.33. 4467H Send message to the display

Condition 3.1.A holds. The message bytes pointed to by HL up to and including a 0DH byte (SOL) or up to but not including a 03H byte (EOM) are sent to the display.

3.34. 446AH Send message to the printer

The same as 4467H except the message is sent to the printer.

3.35. 446DH Convert clock time to HH:MM:SS character format

The current clock value at Model I locations 4041H - 4043H (Model III locations 4217H - 4219H) is converted to HH:MM:SS character format and stored in the 8 bytes pointed to by HL. Registers AF, BC, DE and HL are altered. On exit, HL points to the next byte after the HH:MM:SS field.

3.36. 4470H Convert the date to MM/DD/YY character format

This routine is the same as 446DH, except the date value at Model I locations 4044H - 4046H (Model III locations 421AH - 421CH) is converted to MM/DD/YY format.

3.37. 4473H Insert default name extension into filespec

If the filespec pointed to by register DE has no name extension, insert the 3 characters pointed to by HL as its name extension. The resulting filespec cannot exceed 31 characters. Registers AF and HL are altered.

3.38. 0013H Read a byte from a disk file

This is DOS's single byte read routine even though it starts in ROM. Conditions 3.1.A, B and C hold. If the disk sector containing the NEXT byte of the file is not in the FCB's buffer, it is read into there. The byte is then placed into register A for use by the caller. The FCB's NEXT field is incremented.

3.39. 001BH Write a byte to a disk file

This is DOS's single byte write routine, even though it starts in ROM. Conditions 3.1.A, B and C hold. If the disk sector corresponding to the FCB's NEXT position is not in the FCB's buffer, it is read into the buffer, unless NEXT is on a sector boundary and is equal to EOF. The byte in register A on entry is placed into the buffer, and NEXT is incremented. If the buffer is now full, the sector is written to disk as if a 443CH call.

3.40. 447BH Model III only (performs as Model I 4410H)

For Model III only, performs the same function as call 4410H does for the Model I (see section 3.8). For the Model III, 4410H must not be used.

4. DOS FEATURES

This chapter discusses DEBUG, MINI-DOS, CHAINING, DOS-CALL, JKL and asynchronous execution. DEBUG, DOS-CALL and asynchronous execution are primarily of interest to machine language programmers and those interested in Z-80 code. Other users should make a quick reading of DEBUG and DOS-CALL as they are frequently referred to elsewhere in the manual. MINI-DOS and JKL can be used immediately by everyone. CHAINING can be very complex; novice users will want to test out the chaining concept by using the BASIC program CHAINBLD/BAS to first inspect the sample chain file CHAINTST/JCL and then to create some elementary chain files.

4.1. DEBUG Facility

As an aid primarily for the machine language programmer but also for use by higher level language programmers, NEWDOS/80 has the DEBUG facility for interrupting current execution, inspecting memory, altering memory, inspecting disk, altering disk, single step execution, etc.

DEBUG can be entered in three ways:

1. Simultaneously depressing the three keyboard keys 1, 2 and 3. In order for this 123 action to work the follow conditions must be met.
 1. SYSTEM option AB = N.
 2. SYSTEM option AE = Y.
 3. Either (1) interrupts are enabled or (2) the main program is awaiting keyboard input via the standard keyboard input routine and SYSTEM option AJ = N.
 4. DOS must not be currently using its overlay area (main memory locations 4D00H - 51FFH).
 5. DOS must not have its overlay inhibit enabled.
1. Executing either a RST 30H or a JP 440DH or a CALL 440DH Z-80 instruction.
2. Automatically at, but before, a machine code program commences execution if DEBUG has been turned on via DOS command DEBUG (see section 2.17).

Upon entry, the DEBUG facility will (1) save all registers in the interrupted program's stack, (2) use the next stack locations for its own operations, (3) disable any stops that may have been set on its last exit, (4) display memory using mode and locations as remembered from its last exit, and (5) display the cursor in the lower right hand corner of the display to indicate that the DEBUG facility is awaiting an input command.

All commands, even the single character commands, to the DEBUG facility must terminate with ENTER. If an error is made in keying in a command but before ENTER is depressed, simply backspace over the incorrect characters and type

in the correct ones. If desired, the command may be purged before ENTER by keying shift left arrow.

Both the X and S displays display memory 16 bytes per display line, both in hexadecimal and in character format. If SYSTEM option AI = Y, character formats will include lower case letters.

When DEBUG encounters an error condition, it displays 'ERROR' and waits for the user to acknowledge the error which is done by pressing ENTER to clear the error state.

The DEBUG facility commands are as follows. Wherever numeric values are used, they are always hexadecimal values without the suffixed H unless otherwise specified.

X The DEBUG facility shifts to X display mode, if not already there. The X display contains 15 lines. The 1st through 4th lines contain the 1st 64 byte memory area display. The 5th line displays the interrupted/ replaced contents of Z-80 registers AF, BC, DE and HL. The 6th through 9th lines contain the 2nd 64 byte memory area display. The 10th line contains the interrupted/replaced contents of Z-80 registers AF', BC', DE' and HL'. The 11th through 14th lines contain the Ad 64 byte memory area display. The 15th line contains the interrupted/replaced contents of Z-80 registers PC, SP, IX and IY. The displays for registers AF and AF' also include a bit mask for the associated F register, with an alphabetic character if the bit equals 1 (state set) and a - if the bit equals (state not set). The meanings of the bits (7 - 0) are:

- 7. S = minus sign
- 6. Z = zero
- 5. 1 = unused bit
- 4. H = half-carry
- 3. 1 = unused bit
- 2. P = even parity or overflow
- 1. N = subtraction
- 0. C = carry

Using the X display allows the user to track the registers and three separate memory areas at one time.

S The DEBUG facility shifts to S display mode, if not already there, using X display's 1st memory area's base address rounded down to a 256 byte page boundary as the S display's base address. The S display displays 256 bytes of memory, using 16 display lines.

[n]Daddr1 If in S display mode, the 256 byte block containing addr1 is displayed; if n is specified, the base address of the specified area is changed, but the display won't change since DEBUG is in the S display mode. If in the X display mode, addr1 becomes the base address for the specified area: 1 if n not specified, 2 if n equals 2, and 3 if n equals 3. Examples:

- 1. D7080 displays the contents of locations 7000H - 70FFH if DEBUG is in S display mode. If in X display mode, display area 1 will display the contents of locations 7080H - 70BFH.

2. 3DFFC0 If DEBUG is in X display mode, display area 3 will display the contents of locations FFC0H - FFFFH. If in S mode, the new area 3 address is remembered, but the display is not changed.

[n]; If in S display mode and n not specified, the S display is advanced to the next 256 byte block. If in X display mode, the specified 64 byte display area is advanced 64 bytes: area 1 if n not specified, area 2 if n equals 2, and area 3 if n equals 3.

[n]- If in S display mode and n not specified, the S display is retarded to the next lower 256 byte block. If in X display mode, the specified 64 byte display area is retarded 64 bytes: area 1 if not specified, area 2 if n equals 2 and area 3 if n equals 3.

Haddr1 The DEBUG facility shifts to S display mode, if not already there, displays the 256 byte block containing addr1, enters modify mode and displays a blinking cursor over the hex digit next to be changed. Pressing a key 0 - 9 or A - F causes that hex digit to be replaced in memory and the cursor advanced one position. Pressing right arrow or space advances the cursor one position without memory change. Pressing left arrow retards the cursor one position without memory change. Pressing shift left arrow retards the cursor 4 hex digits without memory change, and pressing shift right arrow advances the cursor 4 hex digits without memory change. Pressing up arrow moves the cursor up one display line without memory change, and pressing down arrow moves the cursor down one line without memory change. The cursor cannot be advanced or retarded outside the current 256 byte page. Pressing ENTER terminates modify mode. Any other key terminates modify mode and raises ERROR state. Example:

M6314 DEBUG is shifted to S mode, if not already there. The contents of 6300H - 63FFH are displayed, and a blinking cursor is displayed over the first hexadecimal digit of byte 6314H. The operator may now key in replacement hexadecimal digits and/or move the cursor around within the displayed 256 byte page.

F[addr1][,hb1][,hb2][,hb3][,hb4] Starting at main memory location addr1, find an occurrence of the specified series of hexadecimal bytes. hb1, hb2, hb3 and hb4 are each 2 hex digits representing a hexadecimal byte. If any of hb1, hb2, hb3 or hb4 are specified, addr1 must also be specified. If none of hb1, hb2, hb3 or hb4 is specified, then the series of hexadecimal bytes last used by an F command is used. If addr1 is not specified, then the memory location +1 of the last F command match is used, thus allowing the user to find successive occurrences of the initially specified byte string. Main memory is searched for an occurrence of the search string of bytes. If found, the address of the first of the matching bytes less 20H is made-the X display's 1st area's base address. This causes the matching byte string to appear at the start of line 3 of the X display. If not found, X display's 1st area's base address is set = 0FFE0H. Example:

F5200,CD,24,44 will start at main memory location 5200H and search for the first occurrence of the three bytes mentioned. Subsequently, the command F will search for the next occurrence of the same three bytes.

If a match takes place in the current stack area, it is possible that the matching bytes will be gone from the stack before they can be displayed, thus causing the user to think DEBUG has stopped erroneously. Further, DEBUG stores the comparison copy of the bytes in the 51xxH region of memory; so if that area is searched, a match will be found upon the compare bytes themselves.

I Execute the interrupted program's current instruction and then re-enter the DEBUG facility. This allows the user to single step execute the interrupted program. The user may then observe the changes (or havoc) wrought by each instruction. Single stepping has some pitfalls however:

1. A full timer interrupt sequence may also execute during the single step.
2. Single stepping is not allowed if the instruction location is less than 5200H or jumps to or returns to a location less than 5200H.
3. The DEBUG facility uses the Z-80 instruction RST 30H to trap for the return to DEBUG after the single instruction has been executed. Therefore, the single stepped instruction should not branch upon itself and should not refer to the next byte following itself as the source or destination of data.

C Performs identical to I except that if the single stepped instruction is a CALL, the entire called routine is executed during the so called single step.

Rdreg,value1 Replaces the interrupted contents of double register dreg with the value value1. Examples:

RDE,C000 replaces the previous contents of register DE with the hexadecimal value C000.

RHL',7100 replaces the previous contents of register HL' with the hexadecimal value 7100.

Ldn1,drsl Relative sector drsl of the diskette mounted on drive dn1 is read into DOS's system sector buffer (Model I locations 4200H - 42FFH; Model III locations 4300H-43FFH). DEBUG then shifts into S mode and displays the sector's contents in that buffer. drsl is a decimal (yes, decimal) value. The user is responsible for providing correct values for dn1 and drsl as DEBUG makes no checks. Once the sector's contents are in the buffer, the user may treat those bytes as normal main memory, may search them using the F command and may alter them by using the M command. However, altering the sector in the buffer does not alter it on the diskette; the WR command must be executed to store the sector back onto the diskette. Since almost all NEWDOS/80 system programs use the system sector buffer for their diskette reads and writes, the user should not use the L or WR commands if the interrupt took place in DOS (in this case the interrupt address is usually below 5200H but be careful of COPY, FORMAT, etc.) and he/she intends to continue the interrupted program's execution.

Warning!!! If passwords are enabled, commands L and WR will be rejected and ERROR state entered. Example:

L1,150 loads the 151st sector of the diskette currently mounted on drive 1 into the system sector buffer.

WRdn1,drsl The contents of the system sector buffer (4200H-42FFH on the Model I; 4300H-43FFH on the Model III) are written to relative sector drsl of the diskette mounted on drive dn1. The parameter definitions and restrictions in the use of command L also apply to command WR. If the specified diskette sector is read protected, it is written read protected.

Warning!!! If you specify the wrong values for dn1 and drsl, you will write the buffer's data to the wrong sector and create for yourself a lot of trouble. Be sure you know what you are doing!!! Example:

WR1,150 writes the current contents of the system sector buffer to the 151st sector of the diskette currently mounted on drive 1.

Q Exit DEBUG to DOS READY. The previous program is forgotten. If the system was in DOS-CALL or MINI-DOS, that state is purged.

G[addr1][,addr2][,addr3] Restore the registers and resume program execution. If addr1 is specified, execution resumes at that location; otherwise it resumes at the memory address specified in the PC register. If addr2 is specified, a breakpoint is set for that location by replacing the byte at that location with the single byte Z-80 instruction RST 30H which when executed will cause the DEBUG facility to be reentered. The replaced byte is not lost (it is restored upon DEBUG re-entry), but it is unavailable during the period from DEBUG exit until DEBUG entry. Addr3 is a 2nd breakpoint address. When addr2 is specified, it is not required that addr1 be specified. Addr2 and addr3 must not be less than 5200H. Examples:

G7000,8400,8425 will set a breakpoint at main memory locations 8400H and 8425H, and will restore the registers and commence program execution at main memory location 7000H.

G will restore the registers and commence program execution at the main memory location saved in the PC register. If the interrupted program was awaiting input (such as DOS READY or BASIC READY) at the time of interrupt, it is still awaiting input. Even though no cursor is re-displayed (as DEBUG does not remember the cursor state), the user may proceed with key input.

4.2. MINI-DOS

There are many times when, during the execution of a main program, the operator would like to interrupt the main program, execute one or more of the DOS library commands and then resume main program execution without any change having occurred to the main program's state during the interruption. NEWDOS/80 provides such a facility, called MINI-DOS.

In order to use MINI-DOS the following conditions must be met:

1. SYSTEM option AB = N.
2. SYSTEM option AF = Y.
3. Either (1) interrupts are enabled or (2) the main program is awaiting keyboard input via the standard keyboard input routine and SYSTEM option AJ = Y.

With these conditions satisfied, the simultaneous depression of the keys D, F and G will cause the main program to be interrupted, its register state saved, and MINI-DOS state to be entered. MINI-NEWDOS/80 READY will be displayed. CAUTION, pressing DFG is not recommended while disk I/O is in progress as a fatal error to the diskette is possible; if exit from MINI-DOS is via MDBORT, then there's no problem.

From MINI-DOS state, the operator may execute any of the DOS library commands except APPEND, CHAIN, COPY and FORMAT. Non-library commands or programs may not be executed under MINI-DOS.

When ready to return to the main program, enter the DOS library command MDRET. If the cursor was displayed before DFG, it will be redisplayed. The main program's register state is restored, and the main program resumes its execution. If the main program was awaiting keyboard record input and a partial record was already inputted, that partial record is still in the buffer even though it is not displayed. If the main program was awaiting keyboard input, whether or not any characters had been entered, upon exit from MINI-DOS, the main program is still waiting. Don't be timid; start keying. If the main program was not awaiting keyboard input, it will go on about its business.

If the main program is not to be resumed, entering the DOS library command MDBORT will terminate both MINI-DOS and the main program, with the system going to normal DOS READY.

Though COPY may not be used under MINI-DOS, simple file copies can be done using DOS library command MDCOPY.

NEWDOS/80 is unable to eliminate all cases where the triple key depression results in one or more of the keys being transmitted as input to the main program. This is especially so when system option AJ = N. These spurious keys usually show up on exit from MINI-DOS. The user should back space over them, and should not use triple key depression when the main program is in text overwrite mode.

As an example of MINI-DOS use, start at DOS READY and execute the following:

```
BASIC
10 PRINT "HELLO": GOTO 10
RUN
```

The BASIC program is now in an endless loop printing the word HELLO on the display. Simultaneously press the D, F and G keys. The BASIC program's execution is interrupted, and the message MINI-NEWDOS/80 READY appears on the display. Now execute the following DOS commands:

```

DIR 0
FREE
CLOCK
CLOCK,N
LIB
SYSTEM,0
PDRIVE,0
MDRET

```

The MDRET command caused the exit from MINI-DOS, and the BASIC program continued execution where it was interrupted. Now, while we have a test program executing, let's try out the entry to DEBUG. Simultaneously depress the 1, 2 and 3 keys. Once again, the BASIC program's execution is interrupted. The DEBUG routine is now active, and the display is loaded with either the X or the S DEBUG display format. Now type in G followed by ENTER. DEBUG is exited, and the BASIC program continues execution. Now, press DFG again to get back into MINI-DOS. Once there, execute DOS command MDBORT. This causes DOS to forget about the interrupted program, to exit MINI-DOS and go to normal DOS READY.

4.3. CHAINING

The DOS commands CHAIN and DO are simply different spellings of the same command; therefore, in this section, only the command word CHAIN will be used where in reality either one can be used.

For most TRS-80 users there are functions which use the same series of DOS commands and/or program responses, and for each of these functions it would save a lot of key stroking, operator time and errors if this keyboard character sequence could be saved in a disk file to be called upon when the operator wishes to execute a specific function.

For example, suppose that each time a reset/power-on is done, the operator keys in the following commands and program responses:

HIMEM,0E800H	Execute DOS command HIMEM
PROGRAM1	Execute program named PROGRAM1
Y	Response to PROGRAM1's 1st query.
50	Response to PROGRAM1's 2nd query.
PROGRAM2	Upon PROGRAM1's completion, execute program PROGRAM2
1	Response to PROGRAM2's 1st query
WORKF1	Response to PROGRAM2's 2nd query
WORKF2	Response to PROGRAM2's 3rd query
BASIC,RUN"BASPGM1/BAS"	Upon PROGRAM2's completion, enter BASIC and run BASIC program BASPGM1.
Y	Response to BASPGM1's 1st query.

Subsequent input to BASPGM1 is assumed to vary from run to run, is therefore not part of the standard sequence and of no concern here. What is of concern is that this same sequence of keyboard input must be keyed in each time.

However, if this keyboard character sequence was placed in a disk file named, for example, XXX/JCL, then this keyboard input sequence can be triggered to occur by executing the DOS command:

```
CHAIN,XXX/JCL
```

The execution of this CHAIN command (see section 2.9) causes keyboard input to come from the file XXX/JCL, starting at the file beginning and transmitting characters as keyboard input when requested by DOS or the executing program. The characters are transmitted upon request until the end of the file is reached, at which time keyboard input is switched back to the normal keyboard. Thus, having keyed in the CHAIN command, the operator may sit back and wait until after BASPGM1 has received its first response instead of having to key in the various commands and responses as needed.

Further, since this keyboard sequence is to be invoked at reset/power-on, the operator may avoid even the keying in of the CHAIN command by setting that command up beforehand as the AUTO command (see section 2.4). This is done by executing the DOS command:

```
AUTO,CHAIN,XXX/JCL
```

Now, when reset/power-on is done, the CHAIN command is automatically executed, and the operator has nothing to do until after program BASPGM1 has received its first response.

Both this process of causing keyboard input to be taken from a disk file and the associated operational mode that NEWDOS/80 is in during that time is called chaining. The files that contain the keyboard character sequences are called chain files.

NEWDOS/80 is not concerned with the creation of chain files; NEWDOS/80 only uses them in response to a CHAIN command (see section 2.9). It is up to the user to decide what keyboard character sequence is to be contained in a chain file, and it is left to the user to build the chain files he/she needs. Probably the simplest way is to use either SCRIPSIT or PENCIL and store the resulting file in ASCII mode. For users that do not have either SCRIPSIT or PENCIL, a BASIC program named CHAINBLD/BAS has been included on the NEWDOS/80 diskette to create and edit simple chain files. To build chain files having other than printable keyboard characters, some other chain file build program must be used.

Chain file creators must remember that, except for any ./ type chaining control records (discussed below), the chain file must contain exactly the keyboard character sequence that DOS or the current executing program expects. Chaining does not guess for you.

During the processing of a chain file, NEWDOS/80 operates in one of two modes, depending upon the setting of SYSTEM option AT.

If SYSTEM option AT = Y, then all requests for keyboard input via the standard keyboard routine are honored from the chain file. This applies to both a request for a record (such as INPUT or LINEINPUT in BASIC) and for a single character (such as INKEY\$ in BASIC).

If SYSTEM option AT = N, then only requests for full records (such as INPUT or LINEINPUT in BASIC) via the standard keyboard routine at ROM location 0040H are honored from the chain file. Requests for a single byte (such as INKEY\$ in BASIC) are honored from the keyboard.

On the NEWDOS/80 Version 2 diskette the user has been provided with (1) the BASIC program CHAINBLD/BAS with which the user can build simple chain files and (2) a sample chain file named CHAINTST/JCL. The instructions for using CHAINBLD/BAS are given in section 6.6. Here, all we want to do is use CHAINBLD/BAS to look at the chain file CHAINTST/JCL. With computer at DOS READY, enter the follow responses:

BASIC RUN "CHAINBLD/BAS:0"	start CHAINBLD/BAS executing
2	chooses file load option
CHAINST/JCL:0	filespec of file to be loaded into memory
L ;	list first page of chain file
;	list next page of file
U	return to edit menu
Q	return to main menu
5	exit from the program

At each step, study carefully what is displayed. This chain file contains a good example of commands, program responses, and chaining control records. **Don't be alarmed at CHAIBLD's 10 second initialization time.** Once you have carefully studied the chain file, exit back to DOS and execute the chain file using the DOS command:

CHAIN,CHAINTST:0

Since most chain character sequences are short, usually less than 100 characters, it is a shame to allocate a full granule of 1280 bytes for each such sequence. Therefore, NEWDOS/80 allows a chain file to be divided into sections with the keyboard character sequence making up each section preceded by a section identification record (see ../0 discussion below) excepting that the first section of a chain file need not have a section ID record. If the chain file section that is to be accessed by a CHAIN command is preceded by a section ID record, the CHAIN command must specify the section ID as well as the file.

During chaining, when either end of file or end of section is encountered, NEWDOS/80 terminates chaining without notification and sets keyboard input back to the normal keyboard routine. This also happens if either DOS command CHNON,N or the chaining ../5N function (see ../ below) is executed. If the current program was awaiting input, the operator will have no indication of this change except that all activity will stop. Usually, the operator knows what will be the first display after chaining terminates; so he/she is ready for it.

If a DOS recognized error occurs during chaining, chaining will be terminated with the message CHAINING ABORT displayed to inform the operator.

If the DOS command CHAIN is executed while chaining, chaining simply forgets the previous file and starts chaining within the new file, which may well be the same file and section as the previous one. CHAIN commands are not nested, and there is no RETURN function in chaining.

DOS-CALL is legal during chaining.

During chaining, there are five ways to alter the sequence of keyboard characters.

1. The current executing program may decide to execute a CHAIN or CHNON command via DOS-CALL (CMD"doscnd" in BASIC).
2. A CHAIN command itself may be part of the chain file. However, for the command to be executed, either DOS must be awaiting its next command or the current program executing must be clever enough to detect the CHAIN command record in its normal record processing and execute the CHAIN command via DOS-CALL (CMD"doscnd" in BASIC).
3. An easier method is by having the chain file contain a ./4 type chaining control record (discussed below) at the point where the change of sequence is to occur. Using the ./4 allows the chaining sequence to be changed regardless of whether DOS or a user program is in control and the sequence change takes place without notification on the display. The limitation of this type of sequence changes is that chaining cannot shift to a different file.
4. The DOS command CHNON (see section 2.10) may be part of a chain file. Remember, DOS must be awaiting its next command. If CHNON,N is specified, chaining is deactivated (though the chain file is not closed and file position is remembered for a subsequent CHNON,Y or CHNON,D command), and keyboard input next comes from the keyboard. If CHNON,Y is specified and DOS-CALL is active, chaining continues but the current DOS-CALL level is exited.
5. A ./5 type chaining control record (defined below) may be used in the chain file instead of DOS command CHNON. The ./5 record function is executed even if DOS is not awaiting its next command.

If the CHAIN command is executed via DOS-CALL (CMD"doscnd" in BASIC), the programmer must remember that DOS remains in DOS-CALL executing DOS commands from the CHAIN file until either end of file, end of section, command CHNON,N or command CHNON,Y (see section 2.10) is encountered. Thus, if a program wishes to activate chaining but wants to process subsequent chain input itself, then the first characters of that chain file or chain file section must be either CHNON,Y or CHNON,N.

Chaining has six control records that may be placed within a chain file. Each of these records must start with either a one character or a 4 character identification sequence and must end with the EOL (ENTER) character. In NEWDOS/80 Version 1, only the one character record identification was used; in Version 2, it is recommended that the four character record identification be used, as the four characters are all printable and thus visible during chain file create or edit. The record ID characters are not displayed during chaining. These control records cause chaining to perform the action described for each. For each special record defined below, the four character record ID is given first followed by the alternative one character ID value.

1. ./0 or one byte = 128 (80 hex). This identifies a section ID record, which must be the first record of a chain section, unless the first section within a file is to be unnamed. The rest of the record is the

section's ID which is used to match against a CHAIN command's section ID, if it specifies one, or against the section ID specified in a ./4 chain control record. Subsequent file characters until EOF or until but not including the next section ID record are all considered part of this new section. Example:

```
./0XXXXXX      identifies subsequent characters as belonging to
chain section XXXXXX.
```

2. ./1 or one byte = 129 (81 hex). This causes the rest of the record to be displayed, and then the system waits for the user to press ENTER before continuing. This is a built in pause function. Example:

```
./1MOUNT WORK DISKETTE      The message MOUNT WORK DISKETTE is
displayed followed by PRESS "ENTER" WHEN READY TO CONTINUE. DOS then
waits for the ENTER.
```

3. ./2 or one byte w 130 (82 hex). The rest of the record is bypassed without further action. This allows the chain file creator/maintainer to place comment records in the file for documentation without them being displayed.

4. ./3 or one byte = 131 (83 hex). The rest of the record is displayed, but no pause is done. This allows the creator/maintainer to display to the operator what is happening. Example:

```
./3PHASE TWO COMPLETED      The message PHASE TWO COMPLETED is
displayed. DOS does not wait but instead continues processing chain
file input.
```

5. ./4 or one byte = 132 (84 hex). The rest of the record is a chain file section ID of 31 characters or less. The current chain file is searched for a chain section whose section ID matches that specified in the ./4 record. When found, chaining continues with the first character of that section. If the section is not found, END OF FILE ENCOUNTERED error is displayed and chaining is aborted. Example:

```
./4xxxxxx      Sequential chain character processing shifts within
the current chain file to the chain section named XXXXXX (see ./0
example above).
```

6. ./5 or the one byte = 133 (85 hex). The rest of the record is either the character Y, N or D. Using this one character parameter, a CHNON function is performed. The advantage of using the ./5 function rather than an actual CHNON command is that DOS does not have to be waiting for its next command. The disadvantage is that the chaining state change is more subtle. The ./5 function is not for the novice. Examples:

1. ./5N chaining is deactivated though the file is not closed.
2. ./5Y chaining remains active but the current DOS-CALL level, if any, is exited.

The novice chain file creator will find it easiest to use none of the chaining control records described above. As experience is gained, try using the `./3` record to display a comment and the `./1` record to display a message and wait for ENTER. Next, try using `./0` records to divide a chain file into sections and then the `./4` record to cause chaining to branch around within a chain file.

The chain file creator/maintainer is responsible for assuring that chaining does not create impossible situations for the system or user programs.

During chaining and if SYSTEM option BC = Y, the operator may terminate chaining by holding down the up arrow key, or the operator may force a chaining pause by holding the right arrow key, and may resume chaining by pressing ENTER.

4.4. DOS-CALL

NEWDOS/80 allows any machine language program to call the DOS routine at 4419H (see section 3.11) to execute a DOS command or user program. This capability is called DOS-CALL. BASIC uses DOS-CALL to execute the `CMD"doscmd"` function.

The calling program builds a DOS command in a buffer and terminates it with a 0DH byte. With HL pointing to the command, the DOS routine at 4419H (see section 3.11) is called to cause DOS to execute the command after moving it to its own buffer and converting lower case to upper.

If the DOS-CALL is executing a user program, DOS does not check for conflict between the calling program and the called program. It is the responsibility of both programs to avoid conflicts. An example of a user program executing under DOS-CALL is the execution of SUPERZAP under BASIC through the `CMD"SUPERZAP"` function.

Furthermore, the registers cannot be used to pass parameters back and forth between the calling and the called programs. On entry to the called program, however, register HL does point to the command parameters. Also, the two bytes at 4403H - 4404H may be used to pass a 2 byte parameter back and forth.

A user program activated under DOS-CALL may itself use DOS-CALL (be careful not to overflow the stack). DOS-CALLS can be nested, with each call activating a new DOS-CALL level.

Upon return from a DOS-CALL, the calling program must check for three states. If Carry is set, an error has occurred that has already been displayed. If the program is to continue execution, then it must decide what to do. If the program is to terminate, it should exit via a jump to 4030H in case this program was itself invoked by DOS-CALL, which will cause an exit to the next higher calling program with C state set.

However, if the returned state is NZ and NC, a DOS error has occurred that has not yet been displayed and the error code is in the right 6 bits of register A (bits 6 and 7 equal 0). If the calling program is to continue operation, it can have the error message displayed by calling 4409H with bit

7 of register A = 1; otherwise it should exit via a jump to 4409H with bit 7 of register A = A This latter action will cause the error message to be displayed and the system goes to DOS READY unless the calling program was itself invoked by DOS-CALL, in which case the error msg will not be displayed and an exit will be taken to the next higher calling program with register A unchanged and NC and NZ states set.

If the returned state is NC and Z, then the called function completed normally. Since all registers except AF are saved at DOS-CALL entry and restored at DOSCALL exit, the only way a parameter may be passed back is by using the two bytes at 4403H and 4404H (17411 and 17412 decimal). Actually, the higher unused bytes of the DOS command buffer, 4318H - 4367H, can be used for communication each way in DOS-CALL, but the programmer must understand that DOS moves all commands into that buffer before executing them.

4.5. JKL

NEWDOS/80 has a small routine for dumping the contents of the display screen to the printer. This feature allows the operator to print information that would otherwise be lost as soon as the display is used for something else.

1. In order to use JKL, the following conditions must be met.
2. System option AD = Y.
3. Either (1) interrupts are enabled or (2) the main program is awaiting keyboard input via the standard keyboard input routine and system option AJ = Y.
4. DOS must not be currently using its overlay area (main memory locations 4D00H - 51FFH).
5. DOS must not have its overlay inhibit enabled.

With these conditions met, the simultaneous depression of the keys J, K and L will cause the main program to be interrupted, its state saved, and the contents of the display dumped to the printer without any editing except that implied by SYSTEM option AX. If the printer is not ready or drops ready, the system will loop waiting for it and no message will be displayed to the operator.

JKL will substitute a period for each display character that is non-printable as defined by SYSTEM option AX.

Pressing the BREAK key will terminate the JKL function, except if the CPU is hung waiting on the printer.

When the dump is completed, the interrupted program is resumed. The problem of spurious input characters discussed in section 4.2 exists here as well.

In earlier versions of NEWDOS, the JKL routine was always resident in main memory. In Version 2, the JKL routine was very reluctantly moved into a system overlay program, thus making it unusable in certain circumstances where it was usable before. For example, JKL can not be invoked from DEBUG.

4.6. Asynchronous Execution

NEWDOS/80, like TRSDOS, allows for a very limited form of asynchronous execution. This is accomplished by inserting a user interrupt routine into DOS's 25ms interrupt chain. The DOS routine (see section 3.8) at Model I location 4410H (Model III location 447BH) must be used to insert the routine into the chain, and the DOS routine 4413H (see section 3.9) must be used to take the routine out of the chain. Refer to these two sections for the required format of the user interrupt routine and how it is invoked.

Again, the user is reminded that the use of user interrupt routines under NEWDOS/80 is incompatible with that under TRSDOS.

5. DOS MODULES, DATA STRUCTURES, AND MISCELLANEOUS INFORMATION

This chapter gives information about the modules on the NEWDOS/80 diskette, about diskette directories and about File Control Blocks. The novice user should read sections 5.1 and 5.4 and leave the other sections for another time.

5.1. Files required on each diskette used with NEWDOS/80

DIR/SYS 2 - 6 granules. Diskette directory. This file is required on every diskette used with NEWDOS/80 as it contains the control information about all files on the diskette. FORMAT or the format part of COPY creates this file automatically, and DOS updates this file as necessary to add, alter, or delete control information about files on that diskette. The structure of the directory is given in section 5.6. Also see section 5.6.2 for correction to HIT sector code for DIR/SYS.

BOOT/SYS 1 granule. Must occupy the first granule of every diskette. On data diskettes this file serves only to reject an attempt to boot using this diskette in drive 0. On system diskettes, the first sector contains the machine code for loading the DOS system from the drive 0 diskette when a power on, reset or jump to location 0 occurs. On NEWDOS/80 system diskettes, the 2nd sector is a duplicate of the first (required for booting on the Model III), and the 3rd sector contains system control information set up by the DOS commands SYSTEM and PDRIVE. FORMAT or the format part of COPY creates this file automatically.

5.2. NEWDOS/80 DOS System Modules

The DOS system consists of 14 program modules which execute from three areas. The resident module SYS0/SYS resides in all the non-data areas from 4000H to 4CFFH. The modules SYS1/SYS through SYS5/SYS, SYS7/SYS through SYS9/SYS and SYS14/SYS through SYS17/SYS all share the DOS overlay area 4D00H - 51FFH (only one module at a time can be in that area). SYS6/SYS executes from both the overlay area and the 5200H - 6FFFH area.

SYS0/SYS 3 granules. DOS's resident module loaded by the bootstrap routine and remains permanently in main memory, except for the DOS initialization routines in the overlay area which are overlaid- when no longer needed. SYS0/SYS handles DOS initialization, disk I/O, clock interrupts, load of other system modules, keyboard intercept, etc.

SYS1/SYS 1 granule. Interrogates DOS commands.

SYS2/SYS 1 granule. Creates files, opens FCBs, allocates file space, allocates FDEs, encodes passwords and loads users programs to be run. Executor for library commands RENAME and LOAD.

SYS3/SYS 1 granule. Closes FCBs, kills files, insert/deletes entries from 25ms chain. Executor for library commands BLINK, BREAK, CLOCK, DEBUG, JKL, LCDVR, LC, VERIFY and most of PURGE.

SYS4/SYS 1 granule. Displays DOS error messages.

SYS5/SYS 1 granule. DEBUG facility.

SYS6/SYS 7 granules. Executes in 4D00H - 6FFFH. Executor for library commands FORMAT, COPY and APPEND.

SYS7/SYS 1 granule. Executor for library commands TIME, DATE, AUTO, ATTRIB, PROT, DUMP, HIMEM and the 1st part of PURGE, SYSTEM and PDRIVE.

SYS8/SYS 1 granule. Executor for library commands DIR and FREE.

SYS9/SYS 1 granule. Executor for library commands BASIC2, BOOT, CHAIN, CHNON, MDCOPY, PAUSE and STMT. Enqueues and dequeues user logical routines and routes each invocation (see DOS routines 4461H and 4464H in chapter 3).

SYS14/SYS 1 granule. Executor for CLEAR, CREATE, ERROR, LIST, PRINT and ROUTE.

SYS15/SYS 1 granule. Executor for FORMS and SETCOM.

SYS16/SYS 1 granule. Executor for most of PDRIVE.

SYS17/SYS 1 granule. Executor for WRDIRP and most of SYSTEM.

5.3. NEWDOS/80 BASIC Modules

NEWDOS/80's Disk BASIC enhancements to the TRS-80's ROM BASIC consists of a main resident module and 8 overlay modules. The modules SYS10/SYS through SYS13/SYS and SYS21/SYS execute from DOS's overlay area, 4D00H - 51FFFH. The modules SYS18/SYS through SYS20/SYS execute from BASIC's overlay area, 5200H - 56FFFH. All of BASIC's modules, except BASIC/CMD, are loaded as needed and must be on the system diskette when needed.

BASIC/CMD 4 granules. Resident module residing in 5700H and up. Executes Disk BASIC's functions. This module need not reside on the system diskette as it may be invoked from a data diskette (like any other program), and once invoked, it is not needed again until BASIC is next invoked.

SYS13/SYS 1 granule. Displays BASIC's error messages and executes 1st part of RENUM. Must be on the system diskette whenever BASIC is active.

SYS12/SYS 1 granule. Executes BASIC direct command REF. Must be on the system diskette if REF will be executed.

SYS11/SYS 1 granule. Executes BASIC direct command RENUM. Must be on the system diskette if RENUM will be executed.

SYS10/SYS 1 granule. Executes BASIC statement's GET and PUT, and must be on the system diskette if either statement is to be executed.

SYS18/SYS 1 granule. BASIC direct statement executor. Must be on the system diskette whenever BASIC is active.

SYS19/SYS 1 granule. Executor for BASIC statements LOAD, RUN, MERGE, SAVE and CMD"F"DELETE. Must be on the system diskette whenever BASIC is active.

SYS20/SYS 1 granule. Executor for a number of disk BASIC statements and usually is the module resident when BASIC is executing a program. Must be on the system diskette whenever BASIC is active.

SYS21/SYS 1 granule. Executor for CMD"O" and must be on the system diskette if CMD"O" will be executed.

5.4. Other Modules on the NEWDOS/80 diskette

DIRCHECK/CMD A program that checks the directory for errors and list or prints the directory contents. See section 6.4.

EDTASM/CMD An editor/assembler for Z-80 code-source and object code from/to disk or tape. See section 6.5.

DISASSEM/CMD A program that disassembles Z-80 machine code. See section 6.2.

LMOFFSET/CMD A program that reads load modules from disk or tape and writes them to disk or tape. The program optionally (1) assigns new load addresses, (2) appends a pre-execution move-program-to-execution-location appendage and (3) prepares the program to run without DOS. See section 6.3.

SUPERZAP/CMD A program that allows inspection and modification of either disk or main memory. Disk operations are diskette or file oriented. See section 6.1.

CHAINST/JCL A sample chain file created by CHAINBLD/BAS.

CHAINBLD/BAS A BASIC program that creates and edits simple record oriented chain files for subsequent use via the DOS commands CHAIN or DO. See section 6.6.

ASPOOL/MAS H. S. Gentry's automatic spooler program as modified by Apparat for NEWDOS/80. See section 6.7.

5.5. Reduced Sized System.

Reduced sized systems can be created, if passwords are disabled, by COPYING the full NEWDOS/80 diskette onto a new diskette and then KILLING the unwanted files. A minimum system to handle open's and close's will consist of 10 granules (BOOT, DIR, SYS0-SYS4). If the DEBUG facility is to be used (including BASIC's CMD"D"), add SYS5. Section 5.2 indicates which additional modules must be added for the various DOS library commands. If BASIC is to be used, section 5.3 indicates which BASIC modules must be added, and section 5.2 indicates which DOS modules must be added if DOS library commands are to be executed via BASIC's CMD"xx" statement.

If the system module loader finds the module's directory entry inactive or encounters an error during loading, then one of the following occurs:

If SYS4 is an active module in the system, then SYSTEM PROGRAM NOT FOUND error will be displayed via a jump to 4409H.

If the jump to SYS4 via 4409H finds SYS4 not in the system, then the Z-80 HALT instruction is executed which on the Model I causes reset and on the Model III stops the computer (the operator must manually press reset).

Modules included in this category are SYS1/SYS thru SYS21/SYS. If any of BASIC overlay modules fail load, the user must carefully execute BASIC to get back the basic program text.

CAUTION!!! Once a system file has been killed from a system diskette, it cannot be restored by simply copying it from another system diskette. The DOS system loader requires that system file FPDEs be in specific FDE slots in the directory and that all of a system file's space be accounted for in the first extent element. Further, SYS0/SYS must occupy the same granules as it did before kill, and it is recommended for efficient system operation that all other system files also occupy the same granules. Once the FPDE has been properly reconstructed, DOS command COPY can then be used to copy the file's contents.

5.6. Diskette Directory Structure

For the Model I, NEWDOS/80 and TRSDOS diskettes are interchangeable provided the NEWDOS/80 diskette's directory consists of only 2 granules (see DDGA parameter of FORMAT, section 2.22, and COPY, section 2.14), and is set up for 10 sectors/track, 2 granules/lump and 5 sectors/granule operations (5 sectors per granule is standard for NEWDOS/80). The files on the diskettes may not be operationally interchangeable between the two systems; system modules, BASIC, ELECTRIC PENCIL, SCRIPSIT, etc., definitely are not though the files they manipulate are.

For the Model III, the directories of NEWDOS/80 and TRSDOS diskettes are NOT compatible; a TRSDOS Model III diskette may not be used directly with NEWDOS/80 and NEWDOS/80 diskettes may not be used directly with TRSDOS Model III. If the NEWDOS/80 single density diskette has a directory of Model I standard position and size, the Model III TRSDOS has a conversion program to

copy the data to a Model III diskette. The COPY function of NEWDOS/80, Version 2, also has a way of copying one, some or all files of a Model III TRSDOS Version 1.3 or higher diskette to or from a NEWDOS/80 diskette (see sections 12.1 and 2.14).

NEWDOS/80 makes all FDE's of a diskette, except those for BOOT/SYS and DIR/SYS, available for use; thus, a 2 granule directory on a newly formatted data diskette has 62 FDEs available. NEWDOS/80 allows the directory to be allocated with up to 6 granules during diskette formatting (see DDGA parameter of PDRIVE, FORMAT and COPY), thereby providing for a maximum of 222 available FDEs.

A diskette's directory always starts on a lump boundary and contains the GAT sector followed by the HIT sector followed by 8, 13, 18, 23 or 28 FDE sectors, depending upon the number of 5 sector granules allocated to the directory (see the DDGA parameter of PDRIVE, FORMAT and COPY). The user is encouraged to study the directory structure by use of program SUPERZAP (see section 6.1). The starting lump number of the directory is always contained as a hexadecimal value in the 3rd byte of each diskette's 1st sector; this value is used by DOS to find the directory.

5.6.1. The GAT (Granule Allocation Table) Sector

The GAT sector is the first sector in the directory and contains the following information:

Granule free/allocated table. Each of relative bytes 00H - 5FH corresponds to a lump and contains the free/allocate status bits for all of that lump's granules. The number of granules per lump is specified by the GPL parameter of PDRIVE and is a value between 2 and 8. The lump's 1st granule's bit is bit 0 (counting from the right), the 2nd granule's bit is bit 1, and so on up to the 8th granule. If the bit equals 0, the granule is free. If the bit equals 1, the granule is allocated or non-existent.

Granule existence table. Relative bytes 60H - BFH correspond to relative bytes 00 - 5FH. If a bit within a byte equals 0, then the corresponding granule for that lump exists and is usable. If the bit equals 1, the corresponding granule does not exist, must not be used and the corresponding bit in 00 - 5FH must equal 1. Actually, though NEWDOS/80 creates these existence bytes during format, it does so only for compatibility with the old style TRSDOS diskettes (where- in these bytes were known as lockout bytes). Actually, NEWDOS/80 never sets a granule non-existent. When necessary, the granule existence table is discarded altogether to make additional GAT sector bytes available to the granule free/allocated table.

In order to maximize the amount of diskette space controlled by the GAT sector, NEWDOS/80 Version 2 allows the free/allocated section of the GAT to extend through, and thereby replace, the existence (or lockout) portion of the GAT. In this case, the free/allocated status bytes are GAT relative bytes 00H through BFH instead of 00H through 5% as discussed above. This extension is automatically done during format if the number of lumps for the diskette exceeds 60H (96 decimal).

The diskette's encoded password is in relative bytes CEH - CFH.

The diskette name is in relative bytes D0H - D7H.

The diskette date is in relative bytes D8H - DFH.

If a system diskette, the AUTO command to be used at reset is contained in relative bytes E0H - FFH. If the first byte of this area is 0DH (EOL), then no AUTO command exists for this system diskette.

5.6.2. The HIT (Hash code Index Table) Sector

The HIT sector is the 2nd sector in the directory. It serves as an index into the FPDEs for the diskette's files and also serves to indicate which FDEs are free and which are in use. If a HIT sector byte equals 0, the corresponding FDE either doesn't exist or is free. If a HIT sector byte is non-zero, the corresponding FDE is in use, and if in use as 'a FPDE, the HIT sector byte's value is a hash code formed from the contents of the FPDE's 6th through 16th bytes (the name and name extension). Thus, when it is necessary to look up a file in the directory, the hash code is computed and the HIT sector searched for a match. If a match is found, the corresponding FDE sector is read and the corresponding FPDE tested for matching name and name extension. If this match fails, the HIT sector search is continued.

The relative position of the HIT byte within the HIT sector is exactly equal to the corresponding FDE's DEC code; for it is by using the DEC code as an index into the HIT sector that the system knows which HIT byte to set non-zero when a FDE is allocated and to set to zero when a FDE is freed.

The HIT sector's 32nd byte is used differently in NEWDOS/80 than all the other HIT sector bytes. This byte contains the count of extra FDE sectors allocated to the directory; the legal values are 0, 5, 10, 15 and 20. This value is set up when the diskette is formatted.

On old Model I diskettes the value of the HIT sector byte for DIR/SYS (2nd byte of the HIT sector) was 2CH which is not the correct value. This incorrect value causes FILE NOT IN DIRECTORY error to appear when the directory file itself is being accessed. For such diskettes, use SUPERZAP to put the correct value of C4H into the HIT sector 2nd byte.

5.6.3. The FDE (File Directory Entry) Sectors.

The rest of the directory's sectors are FDE sectors, with each 256 byte sector containing eight 32 byte FDEs. A FDE is free if bit 4 of its 1st byte equals 0 and in use if the bit equals 1. An in-use FDE is a FPDE if bit 7 of its 1st byte equals 0 and a FXDE if the bit equals 1. When an FDE is freed, only the 4th bit of the 1st byte is zeroed and the corresponding HIT sector byte is zeroed. Nothing else is changed. However, the user may zero the entire 32 bytes of each unused FDE by using the C function of DIRCHECK, thus obtaining a cleaner looking directory.

5.7. FPDE File Primary Directory Entry

Each file, when created, is assigned a directory entry somewhere in the FDE sectors. This entry contains:

1st byte:

Bit 7 = 0.	Indicates FPDE, vice FXDE.
Bit 6 = 1.	If a system file.
Bit 5 = 0.	Undefined.
Bit 4 = 1.	Indicates FDE allocated to a file.
Bit 3 = 1.	If the file has the invisible attribute.
Bits 2 - 0.	Access level code (see PROT parameter of ATTRIB, section 2.3).

2 byte:

Bit 7 = 0. The file may be allocated more space when necessary.
Bit 7 = 1 prohibits this. DIR, ATTRIB, CREATE and the DOS file space allocation routine use this bit.

Bit 6 = 0. The DOS file close function may deallocate any excess granules above the EOF (i.e., apparently not being used by the file).
Bit 6 = 1 prohibits this. DIR, ATTRIB, CREATE and DOS file close use this bit.

Bit 5 = 1. At least one sector of the file has been written to, either new data or updated data, since the last time this bit was set to 0. DIR, ATTRIB, CREATE, PROT, COPY and the DOS sector write routine use this bit.

Bits 4 to 0. Undefined and reserved for future definition.

3rd byte = 0. Currently undefined and reserved for future definition.

4th byte. The lower order byte of the file's EOF. This value is the EOF position within the EOF sector. See FCB 20th byte below.

5th byte. The logical record length (LRECL) (0 = 256) in bytes. When a file is created via a 4420H vector call, the value from register B is stored here. When an existing file is opened, even as a new output file, this value is not updated. This value is never used in NEWDOS/80. The value stored in FCB+9 at open time is that from register B, not from the FPDE.

6th-13th bytes. The file name, padded on right with blanks if necessary.

14th-16th bytes. The file name extension, padded on right with blanks as necessary.

17th-18th bytes. The encode of the update password.

19th-20th bytes. The encode of the access password.

21st byte. The middle order byte of the EOF.

22nd byte. The high order byte of the EOF. The 4th, 21st and 22nd bytes are a 3 byte EOF value. This EOF value, instead of being in RBA format as are the EOF and NEXT fields of the FCB, is maintained in the old TRSDOS format which has the following rules:

 If the lower order byte of the EOF equals 0, the EOF is in RBA format.

 If the lower order EOF byte is not \$, then the EOF value in the FPDE is equal to the actual RBA value plus 255 (the high two byte value of the EOF is incremented by 1).

NEWDOS/80 maintains the directory FPDE EOF field in this manner in order to maintain compatibility with the old Model 1 TRSDOS 2.3 diskettes (see section 12.1). New EOF values for a file are placed into the FPDE only during file-create, write-EOF and DOS close. Thus, if the system fails requiring reset, the user can expect that any file open for output at the time of failure will contain the new data but usually not the new EOF.

See section 12.1 for EOF and NEXT incompatibility with other DOSs.

23-30th bytes. Four 2 byte pairs (extent elements), each specifying a contiguous area of the diskette assigned to this file. The format of an extent element is:

 1st byte:

 255 (0FFH) means the end of the extent elements for this file.

 254 (0FEH) means the next byte contains the DEC for the first or next FXDE assigned to this file.

 0 - 253 (0 - 0FDH) equals the number of the diskette's lump in which the area starts. Other considerations including the number of lumps the GAT sector can handle limit this value to the range 0 - 191. This value is also the relative location within the GAT sector of the byte associated with this lump.

 2nd byte (when the 1st byte is less than 254)

 left 3 bits equals the number of granules (0-7) from the start of the lump to the start of the area.

 right 5 bits equals the number less one of contiguous granules assigned to this area.

31-32nd bytes. An extent element whose 1st byte is either 255 or 254.

5.8. FXDE File Extended Directory Entry

When a file has more than 4 space areas assigned, the additional extent elements are contained in FXDE's assigned to the file. The format of a FXDE is.

1st byte. Bits 7 and 4 are both 1 to indicate a FXDE; all other bits of the byte equal 0.

2nd byte. The DEC for previous FXDE or FPDE of this file. This is a backward chain. The previous entry's 31st byte will be 254, and the 32nd byte will contain the DEC of this FXDE.

Bytes 3-22. Unused and should equal

Bytes 23-32. Are as defined for the FPDE.

5.9. FCB File Control Block

Also known as a DCB (Data Control Block) or an DCB (input/output block).

In order that file information be read from or written to a diskette, a link must be created between that file and the user program. The link is created by the DOS open function (see sections 3.13 and 3.14) and dissolved by the DOS close function (see section 3.15). During the time the link is in existence, the control information for that link is maintained in a 32 byte area of main memory known as a File Control Block. At open time, the user specifies where in user memory this FCB is to be. While this link is in existence, the FCB's area of main memory must not be used for any other purpose. DOS does not remember where the FCBs are. The user informs DOS of which FCB to use for each function that is to use a FCB. Thus, the link is effectively dissolved by simply never using the FCB again in a function call or by using the FCB in the open of a new link. Remember though, if writing to a file where the EOF is being changed, either a DOS close or DOS write-EOF (see section 3.28) function must be done to assure the EOF is properly placed in the FPDE.

At open time (a call to DOS 4420H or 4424H), the caller provides in register DE the address of a 32 byte main memory area for use by the system as a FCB while the file is open. The user must have placed the filespec (terminated by a 0DH or 03H byte) for the desired file into the FCB's 1st bytes, and the DOS close function will attempt to put it back there when done. NEWDOS/80 will accept the Model III TRSDOS 50 bytes area but only uses the first 32 bytes. While the FCB is open, the format for the 32 byte FCB is:

1st byte:

Bit 7 = 1. The link is in existence (i.e., an open has been done).

Bit 7 = 0. The link is not in existence (i.e., either an open has not been done or a close has been subsequently done).

Bits 6-2 = 0. Undefined.

Bit 1 = 1. The value in the FCB's NEXT and EOF fields are RBAs within the diskette, rather than the file. This allows the user to I/O directly to diskette sectors, bypassing the file concept altogether. This bit should never be 1 during byte I/O via the 0013H or 001BH calls.

Bit 0 = 1. Sectors written to the file are written read protected in the same manner as DOS writes directory sectors. This bit should never be 1 during byte I/O via the 0013H or 001BH calls.

2nd byte:

Bit 7 = 1. Either single byte operations or logical record operations (record length in FCB's 10th byte) are being done via this FCB. NEXT value is maintained at the next byte to be read or written. This bit is set to 1 at open time if register B is not 0. It is also set to 1 whenever byte I/O is done via the 0013H or 001BH ROM calls.

Bit 7 = 0. Read and write operations are by full 256 byte sectors with the FCB's NEXT value incremented 256 bytes upon the completion of each successful I/O.

Bit 6 = 0. The FCB's EOF value is to be set equal to the FCB's resulting NEXT value on every successful write operation.

Bit 6 = 1. The FCB's EOF value is to be set equal to the FCB's resulting NEXT value only for those successful write operations resulting in the NEXT value exceeding the current EOF value.

Bit 5 = 0. The FCB's buffer contains the current file sector's data. If bit 5 = 1, the buffer does not contain the current file sector's data; if needed, that sector's data must be read into the buffer.

Bit 4 = 0. The FCB's buffer does not contain updated data not yet sent to the file. If bit 4 = 1, the buffer does contain updated data not yet sent to the file. During DOS close, if this bit is 1, the sector data in the buffer is automatically written to disk. This updated data is also written on every 443FH and 4451H call and on every 4442H, 4445H, 4448H and 444EH call that positions the file within a different sector.

Bit 3 = 1. This FCB is in the NEWDOS/80 Version 2 format for the 18th - 32nd bytes. This bit is set to 1 by DOS open. If bit 3 = the FCB is in the old format and is illegal-in NEWDOS/80 Version 2.

Bits 2 - 0. Access level code (see PROT parameter of ATTRIB, section 2.3).

3rd byte:

Bits 7 - 5. These bits are defined the same as those in the FPDE 2nd byte (see section 5.7). If bit 5 equals 0, the DOS sector write routine sets the bit to 1 in both the FCB and the FPDE just before it actually writes the current sector to disk.

Bits 4 - 0. Undefined and reserved for future definition.

4-5th bytes. The main memory address of the FCB's buffer. The user determines where the buffer is to be and puts this address into register HL before the call to the DOS open routine. Sectors are read from disk into this buffer and written to disk from this buffer.

6th byte. The low order byte of the FCB's NEXT field. This is the relative position within sector value. See discussion for FCB 12th byte below.

7th byte. The relative number of the drive containing the diskette containing the file.

8th byte. The DEC code of file's FPDE. After the FCB is opened, this DEC code is the link between the open FCB and the file's directory information as the FCB itself no longer contains the filespec.

9th byte. The low order byte of EOF. This is the relative position within the EOF sector. See discussion of FCB 14th byte below.

10th byte. The logical record length (LRECL) (0 = 256) for records of this file. This value is supplied in register B by the caller at open time. If not 0 at open time, bit 7 of the FCB's 2nd byte is set to 1, and subsequent DOS sector read or write calls must contain, in register HL, the address of the logical record to be moved to the FCB's buffer (write) or filled from the FCB's buffer (read).

11th byte. Middle order byte of the NEXT field.

12th byte. High order byte of the NEXT field. The 12th, 11th and 5th bytes form a 3 byte RBA within the file of the next byte to be processed, either input or output.

For single byte and logical record I/O, DOS maintains the FCB NEXT field in exact RBA format.

For full sector I/O, DOS also maintains the NEXT field as an exact RBA, but there are subtle actions by DOS that can give trouble if the user is not aware of them. DOS does not change the lower order byte of the NEXT field during full sector I/O. Normally, this byte is zero, and that's fine. However, the user can set this byte non-zero or if the previous I/O done was in single byte or logical record mode the lower order byte will probably be non-zero. The user must be aware of the following rules:

During full sector reads, all three bytes of NEXT participate the EOF check just as for single byte and logical record reads.

During full sector write, when the low order byte of the NEXT field is non-zero, the NEXT field is not advanced 256 bytes upon the successful completion of the write and EOF, if it is updated, assumes that non-advanced NEXT value. The rationale here is that if the NEXT field's lower order byte is zero, the value of NEXT after the successful write is to be at the first byte of the next sector, but if the NEXT field's lower order

byte is non-zero, the value of NEXT after the successful write is to remain within the sector just written.

See section 12.1 for discussion of NEXT and EOF field incompatibility with other DOSS.

13th byte. Middle byte of the EOF field.

14th byte. The 14th, 13th and 8th bytes form 3 byte RBA within the file of the end-of-file (the 1st byte beyond the file's last data byte). This value is initialized from the FPDE at open time, and is updated at sector, logical record or byte write time under control of the FCB 2nd byte, bit 6. See section 12.1 for discussion of NEXT and EOF field incompatibility with other DOSS.

15-22th bytes. Identical to 23-30th bytes of FPDE.

23-24th bytes. For the current FXDE whose 4 extent elements are in the FCB 25th - 32nd bytes, the number in this field represents the relative granule number of that FXDE's 1st extent's 1st granule. If that value equals 0FFFFH, then no FXDE is represented in the 25th-32th bytes.

25-32nd bytes. Identical to 23-30th bytes of the current FXDE, if any.

Discussion of FCB bytes 17-32:

The definition for FCB bytes 17 to 32 has changed from what it was in NEWDOS/80 Version 1 and Model I TRSDOS. It was assumed that very few user programs ever referred to these bytes as they serve only to reduce the number of directory accesses done by the resident DOS. However, some users (such as the old SUPERZAP coded in BASIC) have made use of the old definitions to get around having to open a file when diskette, rather than file, I/O was wanted. NEWDOS/80 Versions 1 and 2 have provided a diskette, as opposed to file, I/O method (see FCB 1st byte, bit 1 definition); that method should be used and those old pseudo FCB methods MUST be discarded to run with NEWDOS/80 Version 2. Failure to do so could be catastrophic; NEWDOS/80 Version 2 has activated bit 3 of FCB 2nd byte in an attempt to head off these bad pseudo FCBs.

This change to the FCB 17-32nd bytes allows the FCB to contain all of a file's extent information for any file having 8 or less extents (DIR with the A option will display how many extents a file has). If the file occupies contiguous diskette space, 8 extents is enough for approximately 300,000 bytes (or 270,000 bytes if the directory is spanned by the file's space).

If the file has more than 8 extents, meaning that more than one directory FXDE is assigned to the file, then the FCB contains space information for the file's 1st 4 extents and the 1 to 4 extents of the FXDE last having a sector read or written. It is quite possible for large randomly accessed files to require a lot more directory accesses than was done under NEWDOS/80, Version 1.

6. ADDITIONAL PROGRAMS SUPPLIED ON NEWDOS/80 DISKETTE

6.1. SUPERZAP

Program SUPERZAP/CMD provides the user with the means to read and write standard 256 byte diskette sectors or any part of main memory, except writing to ROM. Learning to use SUPERZAP is strongly recommended for all NEWDOS/80 owners. If corrections (known as zaps or patches) are to be made to your NEWDOS/80, Apparat will distribute them in written form for application using SUPERZAP. You must know how to use DFS and MODxx. In learning to use SUPERZAP, do your learning on a diskette having data that you can afford to lose!!!!

Certain diskettes are written in non-standard sector formats and are thus inaccessible to SUPERZAP. There exist other programs that read anything that is on a diskette, but do not have some of the other SUPERZAP features. The user, at some time, will probably want to buy one of these other programs from the vendors that sell them.

SUPERZAP operates in both upper and lower case.

Where numeric values are inputted and unless otherwise specified, SUPERZAP assumes DECIMAL unless the value is suffixed with the character H to indicate hexadecimal.

6.1.1. Function Modes

The menu displays the functions available. The user keys in the selected function's characters and then presses ENTER. The SUPERZAP functions are as follows:

DD Display a Disk sector. SUPERZAP will ask for the drive number and the number of the relative sector within the diskette, read the sector and display it.

DM Display a 256 byte page of main memory. SUPERZAP will ask for a memory address, truncate it to a 256 byte boundary and display the page.

DFS Display a File's Sector. SUPERZAP will ask for the file's file-spec. Next, SUPERZAP will ask for the relative sector number within the file and will display that sector.

DTS Display track's sector. SUPERZAP will ask for the drive number, track number and the number of the relative sector on the track. It will then read the sector and display it.

DMDB Display Memory Dump Block. SUPERZAP will ask for the filespec of the memory dump file (created by DUMP, see section 2.20). It will display the dump's base address. Next it will ask for a main memory address within the range of the dump, truncate it to a 256 byte boundary and display the memory page.

VDS Verify Disk Sectors. SUPERZAP will ask if the operator wants a pause when a read protected sector is encountered. Next, SUPERZAP will ask for the drive number and the number of the relative sector on the

diskette of the 1st sector to be verified. Lastly, it will ask for the number of sectors to be verified. It will then proceed with the verify which consists simply of reading each sector within the range specified. When a protected sector is encountered and if a pause was requested, SUPERZAP will display the sector's location and wait for the operator to press ENTER before continuing. VDS is a fast way of finding bad sectors on a diskette that the user suspects have gone bad. While verifying is being done, VDS may be cancelled by pressing up-arrow.

ZDS Zero Disk Sectors. SUPERZAP will ask for the drive number and the number of the relative sector on the diskette of the first sector to be zeroed. Next, it asks for the number of sectors to be zeroed. The zeroing is then done. The read protection status of each sector is not changed.

CDS Copy Disk Sectors. SUPERZAP will ask for the drive number and the number of the relative sector on the diskette of the source (where the data is coming from) range's 1st sector. Next, it will ask for the same data for the destination (where the data is going to) range's 1st sector. Lastly, it will ask the number of sectors to be copied. The copy is then done. Destination sectors are each assigned the read protection status of the corresponding source sector.

CDD Copy Disk Data. This function differs from CDS in that any string of diskette bytes may be copied. SUPERZAP will ask for the drive number and the number of the relative sector on the diskette of the sector containing the source range's 1st byte and then ask for that byte's offset within the sector. It will ask for the same information for the destination range's 1st byte. Lastly, it will ask for the number of bytes (65535 is the maximum allowed) to be copied. The copy is then done. The read protection status of the destination sectors is not changed.

DPWE Display PassWord Encode. SUPERZAP will ask for the password, encode it and display the resulting encode in hexadecimal as it would appear in a directory FPDE.

DNTH Display Name/Type hashcode: SUPERZAP will ask first for the filename and next for the type (name extension). It will then hash them and display the resulting hashcode in hexadecimal as it would appear in the directory HIT sector.

EXIT End SUPERZAP and exit to 440DH (DOS READY).

Since ZDS, CDS and CDD change diskette data, the user is first asked if he/she is sure this function is wanted, just in case the wrong function was keyed.

For CDS and CDD, the copy normally proceeds in ascending byte order for both the source and destination. However, if the highest source byte is within the destination range, the copy is in descending byte order to avoid destructive overlap.

All disk I/O's are done through the normal DOS sector I/O routines. Thus, if an error occurs, system option AM and AW I/O try counts are in effect.

For VDS, ZDS, CDS and CDD, if a disk I/O error results, the operator will be offered the choice of retrying, skipping the sector or terminating the function. In many cases, repeated retrying will eventually work. If the error sector was a source sector, skip will cause the associated destination bytes to receive whatever happens to be in the source's buffer; this should be no problem as the user is faced with a reclaim job anyway.

When SUPERZAP is waiting for a numeric value, keying an X as the value will cause SUPERZAP to terminate the function and return to the menu. If SUPERZAP is waiting for a filespec, a null parameter will terminate the function.

When any of DD, DM, DFS, DTS or DMDB is suffixed with ',P', the sectors or memory pages will be printed as well as displayed. For DD,P, DFS,P or DTS,P, the user will be asked for the number of sectors to be printed. For DM,P or DMDB,P the user will be asked for the number of bytes. If the printer is not ready or drops ready, SUPERZAP will loop waiting on it without operator notification. Pressing the P key will cause printing to pause; press ENTER to continue. Pressing the H key will terminate printing.

6.1.2. Display Mode

For DD, DM, DFS, DTS and DMDB, while a sector or memory page is displayed, SUPERZAP is in the display mode and waits for a display mode command. Except for the F and L commands, the keyed command bytes are not displayed and do not require termination with ENTER; the command is executed as soon as all characters of a display mode command have been keyed. The display mode commands are:

- X** The current function is terminated and SUPERZAP returns to the menu.
- g** Redisplay the same sector or memory page.
- + or ;** Display the next higher sector or memory page.
- Display the next lower sector or memory page.
- J** Restart the same function.
- R** Restart the same function, retaining the 1st parameter unchanged.

SCOPY DD and DTS only. The current sector is to be copied to a specified sector. SUPERZAP will ask for the destination sector's drive number and relative sector number. The destination sector may be the same as the source sector. SUPERZAP will read the destination sector and report its status. Then the source sector's contents are written to the destination sector. SCOPY is useful when a sector is found to have bad parity but, with the exception of a few bytes, is intact; by SCOPYing upon itself, new parity will be generated, and the sector can then be repaired. It is also useful for altering a sector's read protect status.

When SUPERZAP is in the display mode, it has a diskette, file, main memory or memory dump file search capability. The match is on 1 to 4 hexadecimal bytes (without the suffixed H) which are represented by aa,bb,cc,dd. When the search finds a match, the sector or memory block

containing the first byte of the match is displayed with a thin vertical blinking cursor to mark its position. That cursor will disappear as soon as a key is depressed; however, the associated 'find' position is remembered in case the search is to be continued. When SUPERZAP is in display mode, the following commands to perform searching may be keyed in, terminated by ENTER.

F,aa,bb,cc,dd The 1 to 4 hexadecimal match bytes are stored, and the search starts at the first byte of the diskette (if DD or DTS mode) or file (if DFS or DMDB mode) or main memory (if DM mode).

F, The same as above except the previously established match bytes are used.

Fxx,aa,bb,cc,dd The 1 to 4 hexadecimal match bytes are stored, and the search starts within the current sector or block at the xxth relative byte where xx is a 2 digit hexadecimal number without the suffixed H.

Fxx or Fxx, The same as above except the previously established match bytes are used.

F The search continues at the first byte following the position of the first byte of the last match, and the search uses the previously established match bytes.

L,aa,bb,cc,dd This command is to be used instead of F,aa,bb,cc,dd when, in DFS mode, the file being searched is standard load module (i.e., SUPERZAP/CMD, LMOFFSET/CMD, etc.) and the user wants SUPERZAP to purge out all except actual object code bytes from the search. This allows a load module file search for two or more bytes without the imbedded loader control information interfering with the match. The resulting display will still contain the loader control information; the user must be prepared to occasionally see this control information imbedded within the matching bytes. Usually, but not always, this control information is 4 bytes long with the first byte being a hexadecimal 01. Except for purging this control information from the match, L,aa,bb,cc,dd works the same as F,aa,bb,cc,dd. The F command may be used to continue an L type search.

L, The same as above except the previously established match bytes are used.

MODxx DD, DM, DFS and DTS only. SUPERZAP enters modify mode and positions the cursor to the first hex digit of relative byte xx (value 00H - FFH) of the current page or sector.

EXIT End SUPERZAP and exit to 402DH (DOS READY).

If an error occurs during the keying in of a display mode command, the partial command is ignored and the sector or block is redisplayed again.

6.1.3. Modify Mode

SUPERZAP enters modify mode upon execution of the display mode command MODxx.

This mode allows the changing of individual bytes within the current disk sector or memory page. Responses while in modify mode are defined as follows:

Hexadecimal digit character - 9 or A - F. The hex digit at the current cursor position is replaced by the new hex digit, and the cursor is advanced one position. If the cursor wraps around, an error will occur if the next character inputted is a hex digit character. Replacements in a main memory page are for real while replacements in a sector are buffered until the sector is written or a 'Q' command cancels the pending update.

Space or right arrow. The cursor is advanced one position.

Left arrow. The cursor is retarded one position.

Shift right arrow. The cursor is advanced 4 positions.

Shift left arrow. The cursor is retarded 4 positions.

Down arrow. The cursor is advanced one display line.

Up arrow. The cursor is retarded one display line.

ZTxx This sequence is displayed vertically in display column 7 and must terminate with ENTER. All hex digits from and including the cursor position to and including the 2nd hex digit of relative byte xx are zeroed. The cursor is left positioned to the 1st hex digit following relative byte xx, and if wrap around occurs, the next input char may not be a hex digit.

RTxx,jk This command is similar to ZTxx except that each byte's 1st digit is replaced with the hex digit j, and each byte's 2nd digit is replaced with the hex digit k.

Q For sector operations only. Modify mode is terminated, any changes in the buffer are discarded, and SUPERZAP returns to display mode.

ENTER For memory page operations, modify mode is terminated, and SUPERZAP returns to display mode. For sector operations, the operator is asked if he/she really wants to update the sector now. If not, SUPERZAP continues in modify mode. If so, the sector (with any changes) is written back to disk, modify mode is terminated, and SUPERZAP returns to display mode.

When modify mode encounters an error, it will display 'INVALID MODIFICATION MODE CHAR. REPLY '*' TO CONTINUE', Upon receiving * , SUPERZAP returns to modify mode.

6.2. DISASSEM

Program DISASSEM/CMD disassembles Z-80 object code from a standard TRS-80 load module or from main memory. The disassembled code is sent to the display

or to the printer. Generated source text may be sent to disk and a location cross reference may be produced.

Responses to the query 'OBJECT FROM MAIN MEMORY OR DISK?' (M OR D):

1. **null** or **D** Object is a disk load module.

1. Respond to the query 'FILESPEC?' with the filespec of the load module to be disassembled.

2. Respond to the query 'OFFSET OBJECT VIRTUAL ADDRESSES BY? (HEX)' with either null (meaning 0) or a 1 to 4 digit hexadecimal number (without suffixed H) which when added to the load addresses within the load module will give the proper address where the instructions being disassembled would be during normal execution of that code. This parameter is needed when an object module loads to one place in main memory, but actually executes from another. Wraparound is allowed. Example:

If the object module loads into C000H - FFFFH but is to execute in 7000H - AFFFFH, applying an offset of B000 will cause the disassembler to disassemble as if the load was actually done to 7000H - AFFFFH.

3. Respond to the query 'VIRTUAL RESTART LOCATION? (HEX)' with either null (meaning start at the file beginning) or a 1 to 4 digit hexadecimal number (without the suffixed H) which is the listed location of any instruction of the disassembly. This allows restart of a large disassembly within the instruction print portion of the listing, and the location chosen is usually the location value for the first instruction on the page where printing was interrupted.

2. **M** The object code is in main memory.

1. Respond to the query 'OBJECT VIRTUAL BASE ADDRESS? (HEX)' with the 1 to 4 digit hexadecimal location value (without suffixed H) where the object code is considered to execute from, whether or not it is actually there now. In the listing, this value will be the first instruction's printed location value.

2. Respond to the query 'OBJECT REAL BASE ADDRESS (HEX)?' with null (meaning the real and virtual locations are the same) or with the 1-4 digit hexadecimal main memory location (without suffixed H) where the disassembler will actually find the object code.

Responses to the query 'ANY OPTIONS?':

1. **null** No more options to be specified.

2. **PTR** The output is sent to the printer instead of the display.

3. **BFSP** Bypass Full Screen Pauses. In normal operation the disassembler pauses whenever the display screen is full or whenever a break occurs in the sequential locations of the disassembled file. The disassembler waits for (1) ENTER to continue, (2) X to terminate the disassembly or (3) V (object from main memory only) to restart the

disassembly at a new location. The BFSP option bypasses this pausing, causing display to occur as fast as the disassembly can proceed. This option is automatically invoked if option PTR is specified.

The remainder of the options are legal only when the object code is from disk:

4. **NCR** The location reference table is not to be built and no display or listing done of it.
5. **NIP** Do not print or display the disassembled instructions.
6. **STD** Source To Disk The disassembled code is to be sent to disk in the format of an EDTASM source text file. See discussion below.
7. **FGN=xxx** First Generated Name xxx is the 3 alphabetic character name of the first name to be assigned during the STP action described below. The default name is AAA.
8. **RTD** The location reference table is to be stored onto disk. After the reference table is built, the program will ask for the 'REFERENCE TABLE FILESPEC?'. Respond with the filespec of the file to contain the reference table. Reference table files can be used (by a user-created program) to merge the reference tables of two or more programs. See below for file format.
9. **REA** Enable listing of all types of references; this is the default.
10. **RE&** Enable list of the specified reference type where '&' is one of L, P, R, S, T, U, V, W or X. Reference types are defined at the beginning of each location table listing.
11. **RIA** Disable list of all types of references.
12. **RI&** Disable listing of the specified reference type where is one of L, P, R, S, T, U, V, W or X.

The disassembler operates through four phases:

1. If object code from disk and option NCR not specified, DISASSEM displays 'BUILDING CROSS REFERENCE TABLE' and passes through the object code building the location reference table. For a large disassembly this will take some time. If insufficient main memory for the table, the disassembly will terminate.
2. If RTD option specified, this phase writes the location reference table to disk.
3. List disassembled instructions to display or printer. If STD specified, the resulting text is also written to disk. On the disassembled instruction print lines, column 1 indicates the number of references to bytes of the instruction; the value is hexadecimal with blank meaning and F meaning 15 or more references. Column 2 indicates which bytes of the instruction have been referenced. If blank and column 1 non-blank, then only the instruction's 1st byte is referenced; otherwise

the hex digit represents a 4 bit binary mask of which bytes, from the left, are referenced.

4. If object is from disk and NCR is not specified, the location reference-table is displayed or printed. The definitions of the reference type codes are given first. Then, in ascending numeric order, every referenced location is listed with the location of every referencing instruction. Suffixed to each referencing location value is the reference type code for the Z-80 instruction making the reference.

If the disassembler finds something wrong with the object module, either 'DISK OBJECT FILE FORMAT NOT AS EXPECTED' or 'PAST END OF FILE' will be displayed and the disassembly will terminate.

While the disassembled instructions are being displayed or printed, holding down P will cause a pause; press ENTER to continue. Holding down X will terminate the disassembly. At most other times when DISASSEM is awaiting a user response, the disassembly may be terminated by holding down up-arrow and pressing ENTER.

For main memory disassemblies, the operator may shift the disassembly point at will. When the disassembly is paused, keying V will display the query 'VIRTUAL RESTART LOCATION? (HEX)'. The operator responds a 1 to 4 hexadecimal digit value, which is the main memory location where the disassembly is to restart.

If the PTR option is specified and after all options have been specified, the following occurs:

Respond to the query Q LINES PER PAGE, EXCLUDING TOP AND BOTTOM MARGINS? (1-255)' with the number of printable lines per page.

Respond to the query '# LINES EACH FOR TOP AND BOTTOM MARGIN? (0-10)' with the number of lines the disassembler is to skip at both the top and bottom of each page. If 0, the disassembler does no paging action. What the disassembler does for top and bottom margins is completely independent and in addition to anything a printer driver may be doing.

Respond ENTER to the query 'REPLY "ENTER" WHEN PRINTER AT TOP OF PAGE' when the printer is on and at top of page.

Respond to the query 'HIGH ASCII CODE FOR PRINTER? (5A - 7F)' with the 2 hexadecimal digit value (between 5AH and 7FH) for the highest printer code for your printer.

The STD option causes the disassembled code to be converted into EDTASM type source text code. The resulting STD output (if not too large) can be loaded and assembled by EDTASM. The outputting of source text via the STD option works as follows:

After the cross reference table build phase and the RTD phase, respond to the query 'ASSEMBLER SOURCE TEXT OUTPUT FILESPEC?' with the filespec of the file to contain this generated source code. The file will be opened, and the generated text sent to it during the main disassembly phase.

All numeric values within the disassembled code are replaced with a 3 character alphabetic name unique to that value. The names are assigned arbitrarily in ascending alphabetic order with the first name assigned either AAA or the name specified by the FGN option.

If a numeric value does correspond to a disassembled location, the name assigned to that value is placed in the location name field of that location's instruction when it is sent to disk and displayed or printed.

If a numeric value does not correspond to a disassembled location, an EQU statement is generated at the end of the source text to equate the name with the value.

ORG statements are generated as necessary, and the END statement is generated as the last text statement.

The format of the reference table file created by the RTD option is:

1. 1 byte = C0H. Backward EOF. Ignore it.
2. 1 or more entries of the form:
 1. 2 byte memory location value, 1st byte = low value, 2nd = high.
 2. Control byte, bits 7 - 0 (7 is left most):
 - 7-6 = 11. Dummy last entry in table. Ignore all other bits and bytes of the entry.
 - 7-6 = 01. Referencee entry. Bits 5-0 = 0. The location is referenced by one or more of the subsequent referencer entries.
 - 7-6 = 00. Referencor entry. The instruction at this location referenced the location of the previous reference entry. Bits 5-0 contain the references instruction type: 0 = S, 1 = T, 2 = U, 3 = V, 4 = W, 5 = X, 8 = P, 9 = L, and 10 = R. See a reference listing for definitions.

6.3. LMOFFSET

Program LMOFFSET/CMD reads a tape or disk load module, displays its load information, optionally changes the program's load area, optionally attaches an appendage enabling the program at execution time to move itself from its load area to its execution area, optionally prepares the module to run under non-disk BASIC via SYSTEM, and stores the module onto disk or tape with a new name.

LMOFFSET functions as follows:

1. Reads either a tape-type assembly load module from tape or a disk-type assembly load module from disk.

If from disk, LMOFFSET asks for the source filespec.

When reading from tape, a single * will be displayed when LMOFFSET is ready for the tape. Do rewind (if necessary) fast forward positioning (if necessary) and press PLAY. *** appears when tape read synchronization has completed. The character C will be displayed when a bad checksum is encountered. The character P will be displayed if leading extraneous data bytes encountered. The character I will be displayed if imbedded extraneous bytes are encountered.

2. Displays (1) the area into which the module will load, (2) possible conflicts with system storage and (3) the module entry point. If an appendage is scheduled to be applied, the entry point will be into the appendage.

3. Asks for a new load point. Reply either with a new load point or simply reply ENTER if satisfied with the current load point. If the user is simply transferring the load module without change, respond ENTER to the first request for a new load point and LMOFFSET will go directly to step 7 below.

4. If a new load point specified, LMOFFSET asks if the appendage is to be suppressed.

If the appendage is to be suppressed, the resulting module can only be used via the DOS library command LOAD as there is no appendage to move the program to its execution area and the entry point is forced equal to 0. The resulting output load module can be used via LOAD where two or more load modules are loaded into main memory and then stored as one load module via DOS library command DUMP.

If the appendage is not to be suppressed, then LMOFFSET will append to the user program either a DOS enabled appendage or a DOS disabled appendage, depending on whether DOS is to be disabled or not.

5. If a new load point was specified, LMOFFSET goes back to 3 above to display the resulting load information and ask for a new load point. If another load point is given, it cancels the one specified earlier, including its scheduled appendage, if any.

6. Finally, when the response to 3 above is a null, then if a new load point was specified and the appendage is not suppressed, LMOFFSET asks if DOS is to be disabled. If so, the DOS disabled appendage is selected; if not, the DOS enabled appendage is selected.

7. LMOFFSET next asks if the destination is disk or tape.

If the destination is disk, LMOFFSET asks for the filespec of the load module file to be created.

If the destination is to tape, LMOFFSET asks for the tape module name and then which tape speed (L or H). Next it asks for ENTER when the tape is positioned and in record mode.

8. The resulting load module is then written to disk or tape. If a new load point was specified, (1) the load address for each object code record is altered, (2) if the appendage was not suppressed, an extra

object code record (the appendage) is inserted before the entry point record and the entry point is set to the appendage's 1st byte, and (3) the entry point is set to 0000 if a new load address was specified and the appendage was suppressed.

9. When the destination file write is completed or if an error or other type of termination occurs during step 7 or 8 above, LMOFFSET asks if the same module is to be written to another file (which may be the same file). If so, steps 7 and 8 above are repeated.

10. When all done or if an error or other type of termination occurs while not in steps 7 or 8, LMOFFSET asks if another source load module is to be processed. If so, execution returns to step 1 above; if not, LMOFFSET exits back to DOS.

The up-arrow key may be used at any time to terminate the current LMOFFSET function. If LMOFFSET is waiting for a response, hold down the up-arrow key and press ENTER.

A module can end up with multiple appendages if the output from one LMOFFSET run is made the input to another, but doing this is strongly discouraged; in the case where one appendage is a DOS disable appendage, it must never be done. LMOFFSET knows nothing of a previously existing appendage appended by a previous execution of LMOFFSET.

LMOFFSET does not perform any object code relocation!!!! It only assigns code to new load locations so that DOS can load the module from disk without damage to DOS.

If the source program loads into the display area (3C00H - 3FFFH) without overflowing it, those object code records will not have their load addresses modified.

The appendage added to a module by LMOFFSET starts with 64 bytes of zeroes. This area is available to users to patch in special code. The load address of this patch area is the same as the module's resulting entry address, providing there is only one appendage. Z-80 code patched into this area will be the first executed when that program commences execution. This will be done before the program is moved to its execution locations and before DOS is disabled, if DOS is to be disabled.

When a program is to run in any part of the DOS area, a DOS disabling appendage must be specified. The DOS disabling appendage causes the user program to execute as if it was loaded from tape under the non-disk BASIC SYSTEM function.

When the resulting user program module is executed, the action is as follows:

For a DOS enabled appendage:

1. Executes any user supplied code in the 64 byte patch area.
2. Moves the main program to its execution locations.
3. Commences execution of the main program.

For a DOS disable appendage:

1. Executes any user supplied code in the 64 byte patch area.
2. Moves the display screen contents to high memory.
3. Displays the following:

RECORD AND THEN PERFORM THE FOLLOWING INSTRUCTIONS

1. HOLD DOWN BREAK KEY AND PRESS RESET TO ACTIVATE NON-DISK BASIC.
 2. RELEASE BREAK KEY AND ENTER BASIC INITIALIZATION RESPONSES.
 3. ENTER "SYSTEM".
 4. ENTER ".".
4. When the operator has done the above, the appendage continues execution.
 5. Restores the screen contents from high memory.
 6. Moves the main program to its execution locations.
 7. Commences execution of the main program.

6.4. DIRCHECK

The DIRCHECK/CMD module tests and lists the target diskette's directory. If errors are found in checking the directory, they are listed before the directory listing. DIRCHECK also allows the option of cleaning up (not repairing) the directory, and, as an aid to moving single density diskettes back and forth between the Models I and III under NEWDOS/80, allows the option of writing the directory protected.

To the query 'OUTPUT TO PRINTER', reply Y if output to go to printer and N if to go to the display.

To the query 'WHICH DRIVE CONTAINS TARGET DISKETTE', reply the target drive number, in decimal.

DIRCHECK reads the BOOT sector (the diskette's 1st sector), and tests that the first 2 bytes are 00H and FEH respectively. If they are, DIRCHECK uses the 3rd byte as the number of the lump at whose first sector the directory starts. If the first 2 bytes are not correct, DIRCHECK displays '***** DISKETTE 1ST SECTOR NOT "BOOT". ASSUMING DIRECTORY STARTS ON LUMP 17 DECIMAL.'.

DIRCHECK proceeds to read the directory. In previous NEWDOS versions, DIRCHECK refused to process a directory that was not write protected. Because of the problem of moving single density diskettes between the Model I and Model III under NEWDOS/80, an unprotected directory will now be accepted, with two error messages displayed, one at this time and one after the files have been listed. The error message is '***** AT LEAST ONE DIRECTORY SECTOR UNPROTECTED'. If this message appears along with many other errors, the user

can assume that DIRCHECK has not found the directory and should NOT execute the W function described later.

DIRCHECK uses the drive's PDRIVE (see section 2.37) data to determine the number of lumps and granules accounted for by the directory. If the PDRIVE data is not correct for the diskette, it is very probable DIRCHECK will list errors that are, not actually present.

Complaints, if any, about the directory are next listed. If a number is given, it is in hexadecimal for use in directory repair via SUPERZAP. Do not try to repair a bad directory unless you know what you are doing!!!!!! The next best thing is to try to extract valued files via COPY and then re-format the diskette having the bad directory.

If the complaint is about a directory entry for a file, either the primary or an extended entry, the hexadecimal code is the DEC for the file's FPDE. When the complaint deals with a file extended directory entry but does not specify the file name/type, the hexadecimal code is the DEC for the FXDE itself. When the complaint deals with a HIT sector byte, the hexadecimal code is the relative location of that byte in the HIT sector. When the complaint deals with a GAT sector byte, the hexadecimal code is the relative location of that byte in the GAT sector. When the complaint deals with a granule, the hexadecimal value is expressed in bb,r format where bb is both the lump number and the relative byte location of the lump's byte within the GAT sector and x is both the relative granule within the lump and the bit number, counting from zero from the right, within that GAT byte.

The diskette's name and date are next listed.

The files are next listed, with numeric values in decimal and the following definitions:

S System file.

I File has invisible attribute.

P=nnn File has access level nnn, and both update and access passwords are non-blank.

EOF=sss/bbb End Of File value. ass = the relative sector within the file. bbb = the relative byte within the sector.

nnn EXTS nnn is the number of extent elements, maximum of four per FDE, used to account for this file's disk space.

nnn SECTORS The number of sectors allocated to this file.

Lastly, the number of free granules and locked out granules for the diskette are displayed. If the diskette contains more than 60H (96 decimal) lumps or if GAT relative byte 60H equals 0FFH, DIRCHECK assumes that there is no lock-out (existence) table. Note, NEWDOS/80 does not mark granules as locked out; the lockout table is maintained only for compatibility with Model I TRSDOS.

If at least one directory sector is unprotected, another error message indicating such is displayed.

'FUNCTION COMPLETED' message is displayed followed by the query:

```
REPLY
N   TO EXIT PROGRAM
Y   IF ANOTHER DISKETTE FOR SAME SPECS
I   FOR PROGRAM RE-INITIALIZATION
W   TO WRITE DIRECTORY SECTORS PROTECTED
C   TO CLEAN UP (NOT REPAIR) THE DIRECTORY
```

Reply with one of the following:

N Program exits to DOS at 402DH.

Y Another diskette to be checked but with same response to the printer query.

I Another diskette to be checked but with different response to the printer query.

W The directory sectors are read and re-written in protected state. Refer to specifications for DOS command WRDIRP (section 2.49) and option SYSTEM option BN (section 2.46). This function is only meaningful for single density diskettes that are going from Model I to Model III or vice versa or used interchangeably.

C All unused FDEs within the directory are zeroed. This is a cosmetic function only that clears out residual information from no longer used FDEs. Normally, when DOS releases FDEs via KILL or automatic space deallocation, it only zeroes bit 4 of the first byte of the FDE, leaving the rest of the information for the remote possibility that the sophisticated user will attempt to reclaim the file or the sectors it used to own.

During display or printing, pressing:

BREAK - processing will pause at end of current line or line group.

ENTER - continues processing.

UP-ARROW - terminates displaying or printing.

6.5. EDTASM Disk Oriented Editor/Assembler

35 months ago Apparat converted the TRS-80's tape oriented editor/assembler to:

1. Read text from disk as well as cassette.
2. Write text and/or object to disk as well as cassette. Disk files are validity read after all sectors written.
3. Allow down-arrow scrolling to display up to 15 text lines.

4. Prevent the confusing printer output associated with DEEM. Only the 1st byte of associated object code is listed.
5. List symbols in alphabetical order with reference list.
6. Accept and convert lower case alpha to upper.

It was anticipated that Radio Shack would soon come out with a disk oriented editor/assembler that would eliminate any need for the Apparat enhancements. To a degree that has come to pass, but not sufficiently to bury the Apparat enhanced version. Since the Apparat enhanced version is based on the copyrighted tape editor/assembler, Apparat has always required and still requires, as a pre-condition of use of its enhanced version, that the user purchase a copy of the TRS-80 tape editor/assembler and thereby pay the royalty due. In an effort to enforce this, Apparat has always refused, and will continue to refuse, to supply any documentation for the editor/assembler beyond that dealing explicitly with Apparat's enhancements.

This EDTASM is essentially the same as that offered with NEWDOS/21 and NEWDOS/80 Version 1 except:

1. EDTASM will now display, as part of the 'A' CMD, after the TOTAL ERRORS display, the number of bytes left in the text area so the user can judge his approach to symbol table overflow or text buffer overflow.
2. (Model III only) Object code cannot be outputted to tape. The user must output the object code to disk and then use LMOFFSET to copy it to tape.

Supplemental instructions for the editor-assembler.

1. To load a text module into the text buffer, enter one of the following commands:

1. L D=filespec1 if text from disk
2. L T=nnnnnn if text from cassette

where filespec1 is the filespec for the assembler text module to be loaded into the text buffer from disk and nnnnnn is the name of the assembler text module to be loaded into the text buffer from tape.
Examples:

1. L D=OLDTEXT/SRC:1 loads the assembler text file OLDTEXT/SRC into the text buffer from the diskette currently mounted on drive 1.
2. L T=OLDTXT loads the assembler text file OLDTXT into the text buffer from tape.

If the text buffer already contains text, the query 'TEXT IN BUFFER. ARE YOU CONCATENATING???' appears. If you are not concatenating, reply N; the buffer is marked empty before loading the specified text module. If you are concatenating, reply Y to cause the new text to be appended onto the end of the old. No concern is shown for overlapping sequence numbers; therefore you should execute a N

EDTASM command upon completion of the load to assure a valid set of ascending sequence numbers.

2. To store a text module:

- 1. W D=filespec2 if text going to disk
- 2. W T=nnnnnn if text going to cassette

where filespec 2 is the filespec of the disk file to receive the assembler text from the buffer and nnnnnn is the one to six character name given to the text file written to tape. Examples:

1. W D=NEWTEXT/SRC:1 The assembler text (not the object code) currently in the text buffer is written to file NEWTEXT/SRC on the current diskette mounted on drive 1.

2. W T=NEWTXT The assembler text currently in the text buffer is written to tape and named NEWTXT.

3. For A commands with NO option not specified, respond to the query 'OBJECT FILE TO DISK OR TAPE? REPLY D OR T?':

1. T (Model I only) Object code going to cassette. The program name will come from the A command.

2. D Object code going to disk. Respond to the query 'OBJECT FILESPEC?' with the nnnnnnnn/ttt.pppppppp:d filespec of the object module. The file will be opened immediately, but not written until end of assembly listing. The name in the A command is ignored.

4. When an output text or object disk file is opened, one of the following is displayed:

1. 'FILE ALREADY EXISTS. USE IT????'. Reply Y if this is your intention. Otherwise reply BREAK to terminate the W or A command.

2. '***** FILE NON-EXISTENT. REPLY 'C' TO CREATE IT'. Reply C if this is your intention. Otherwise reply BREAK to terminate the W or A command.

5. Due to an error in the original DOS, EDTASM runs with interrupts disabled (except when re-enabled by disk I/O) in order that use of BREAK will function properly.

6. This EDTASM can execute in a regular TRSDOS Model I environment.

7. This EDTASM uses the standard keyboard, display and printer routines and control blocks. Users altering the system beware!!!!

6.6. CRAINBLD

The BASIC program CHAINBLD/BAS is a simple program to allow users to create and modify chain files (chaining is discussed in section 4.3).

CHAINBLD operates in record mode, requiring that an EOL character (ENTER character) appear in the file at least every 240 bytes, and it treats each occurrence of the EOL character as both the end of a BASIC input line and the end of a record within a chain file. All inserts, deletions, replacements, moves and copies are done in terms of records.

Furthermore, CHAINBLD makes no provision (except for the old Version 1 hex codes 80 - 83) for the file to contain special non-printable characters. The rule is that if the string resulting from the BASIC statement LINEINPUT C\$ does not contain a given character, then that character cannot become part of the chain file. The exception is the EOL character, which is automatically supplied by CHAINBLD. If the user needs special characters in his/her chain file, some other program must be used to build the chain file. As a last resort, there is always SUPERZAP.

The CHAINBLD program starts off with a 16 second initialization period while it allocates maximum space to the string area. Users are warned that if BREAK is used to interrupt or terminate the CHAINBLD program, they must remember that all available space has been assigned to the string area and that due to this lack of space, some functions will not work. If a CLEAR is done to free up some space, be sure to specify .a string area size.

After initialization, the main menu is displayed (not to be confused with the edit menu). The choices are:

1. DELETE ALL TEXT LINES All the text lines in the string area are deleted and the edit menu is displayed. When CHAINBLD starts execution, there are no text lines in the string area.
2. LOAD EXISTING TEXT FROM DISK Use this option to edit an existing chain file. If the string area already contains text lines, CHAINBLD will ask if those lines are to be deleted. If not, CHAINBLD returns to the main menu as it assumes the user wants to do more with the previous text. Otherwise the old text lines are deleted.

CHAINBLD will then ask for the existing chain file's filespec. If the filespec does not contain a name extension, the name extension JCL is assumed. The file is then loaded into the string area. The file cannot exceed the string area capacity and cannot have more than 1000 lines. The file must be segmented into records as discussed above. After the load, CHAINBLD displays the edit menu.

3. SAVE TEXT TO DISK The user has completed the creation and/or editing of the chain file text and now wants to write it to disk. If there are no text lines, the CHAINBLD will ask if a null file is to be written; if not, CHAINBLD goes back to the main menu.

Next, CHAINBLD asks if the file is to be written so that it can be processed by NEWDOS/80 Version 1. If so, any ./0 through ./3 chain control records are changed as they are outputted by substituting the corresponding single byte control code (80H - 83H) in place of the ./x character sequence. The text in the string area is not changed.

CHAINBLD then asks for the output file filespec. If the filespec does not contain a name extension, the name extension JCL is used. The file is then written to disk. When done, CHAINBLD goes back to the main menu.

4. EDIT TEXT This option does nothing except display the edit menu.
5. EXIT PROGRAM If the string area contains text that has not yet been written to disk, CHAINBLD asks if the user really wants to exit the program; if not, CHAINBLD goes back to the main menu. Otherwise CHAINBLD deletes all text lines and releases all string space except 50 bytes. The program then ends in the normal manner.

When the edit menu is displayed the user has a number of choices:

1. List text lines. The text lines are implicitly numbered in sequential order regardless of the changes that take place in the text. Line numbers do not belong to individual text lines. Instead a line number indicates the line's position at the current time within the file. This means that insert, delete, copy and move all change the line numbers of some or all of the text lines. The L and ; edit commands allow the user to display the text lines. L; displays the first line. L/ displays the last. L52 displays the 52nd line. In each case, if any text lines follow the target line in the text, they are also displayed. The ; edit command allows forward text paging.
2. The I edit command allows for a one or more text lines to be inserted in the text after the specified line. 10 does inserting at the start of the text. I/ does inserting at the end of the text. 123 does inserting after line 23. Lines are inserted into the text until, but not including, a line containing the //. character sequence is encountered. That character sequence terminates the line insert mode.
3. The R edit command allows a new line to replace an old line. R43 causes text line 43 to be replaced with the new line that CHAINBLD will ask for.
4. The D edit command allows one or more text lines to be deleted. D34 deletes text line 34. D 20 41 deletes text lines 20 through 41.
5. The X edit command allows the specified text line to be added onto. Note that CHAINBLD does not actually allow a line to be edited. The edit mode really refers to editing the entire text.
6. The C edit command allows the specified lines to be duplicated to another part of the text. C 20 30 5 causes a copy of text lines 20 through 30 to be inserted after text line 5. Please note that the old lines 20 through 40 will now have line numbers 31 through 42.
7. The M edit command allows the specified lines to be moved to another position in the text. M 20 30 5 causes the text lines 20 through 30 to be deleted from the text and reinserted after text line 5.
8. The U edit command redisplay the edit menu.
9. The Q edit command redisplay the main menu.

The best way to learn CHAINBLD is to use it. The NEWDOS/80 distribution diskette comes with a sample chain file named CHAINTST/JCL. Load it in and look at it. Once in the string area, you may modify the text as desired, but do not store it back out as CHAINTST/JCL; use some other name.

6.7. ASPOOL

1. The object module ASPOOL contained on the NEWDOS/80 diskette is H. S. Gentry's automatic Spooler Program, modified by Apparat to operate with NEWDOS/80 and to self-relocate. This program will automatically direct your printer output to the disk, and then automatically print it on the printer. This spooler program will print in the background while your foreground main program is executing provided the main program every second or so either sends a byte to be spooled or checks the keyboard for a new input character.

This spooler program is included on the NEWDOS/80 diskette as a free program to NEWDOS/80 owners. It is NOT a fully supported part of NEWDOS/80.

The basic operation of NEWDOS/80 DOS assumes that output that DOS sends to the printer will not involve disk I/O enroute to the printer. Therefore, the spooler discards all printer output it senses coming from DOS (such as PRINT, JKL, DIR with P option) with the warning message CAN'T SPOOL FROM DOS being displayed once for each spooled file.

This spooler program does NOT allow a spool file to be printed multiple times; once printed, the file EOF is set to 0 and the file closed to reclaim the file space. This spooler program does NOT remember spool contents from one spool activation to the next (this includes a reset). The user is warned that while the spooler is active, do NOT use reset or DOS library command BOOT to get to DOS ready. Instead, if another way is not available, use DFG to get to MINIDOS and then DOS library command MDBORT to get to DOS READY or use '123' to get to the DEBUG facility and then use DEBUG command Q to get to DOS READY.

2. INITIAL SETUP. Create a working spool module.

Before the spool system can be used, working program module copy(s) of ASPOOL must be set up. You should set up a working program module for each different configuration you intend to use. When making a working program module, the input module 'filespec1' must ALWAYS be ASPOOL/MAS or a copy of it, and the output module 'filespec2' must NEVER be ASPOOL/MAS. To create a working spool program module (as opposed to the master), enter the DOS command filespec1,I (example: ASPOOL/MAS:0,I). The program will then ask for parameter specifications:

The program asks if the software printer driver whose address in is 4026H - 4027H at the time of spooler activation is to be used to drive the printer. Reply Y for yes or N for no (the spooler will drive the printer). If N, then:

The program asks if the printer is parallel or serial. Answer P for parallel or S for serial. If serial, then:

The program asks if the printer is an H14 type. Respond Y for yes and N for no.

The program asks if the printer output is to be formed into pages with a form feed between pages. Reply Y for yes and N for no. If Y, then:

The user will be asked for the number of print lines per page. Enter a number between 10 and 99.

The program asks if the printer uses a soft or hard form feed. A soft form feed is done by counting the number of lines printed and then printing carriage returns (OUR) (with or without line feeds (OAR)) until the end of the page is reached. A hard form feed is a single control character that causes a form feed function. If your printer will recognize a hard form feed answer H, otherwise answer S. If soft form, then:

The program asks for the total number of lines per page. Answer with a number between 10 and 99.

The program asks if a form feed is to be done at the end of each print file. Reply Y for yes and N for no.

The next question concerns automatic linefeed on each carriage return. Some printers linefeed on carriage returns and the computer should not output linefeeds. If your printer is of this type (Radio Shack standard) answer the question with N. If you want the software to generate linefeeds then answer with Y.

The program asks for the number of the disk drive that will be used to spool the print data. Answer with a number from 0 to 3.

The program asks for the number of seconds to transpire after the last keyboard key inputted until the spool program can start printing again. Respond with a 2 digit value 00 - 59. The purpose of a non-zero delay is to allow the keyboard to have primacy over the printer. When a keyboard key is depressed and if the spool program is printing a file, printer action will pause while keys are being inputted and until the required number of seconds have passed since the last key.

The program asks if the printer is to be driven by the timer interrupts (every 25ms on the Model I; every 33 or 25ms on the Model III) as well as via keyboard input and spooler output. Reply Y for yes if the interrupts are to be used; reply N for no. Allowing the interrupts to be used enables the spooler program to print while a foreground program is executing that does not frequently check the keyboard or send output to the spooler. The disadvantage of using the interrupts is that for a buffered printer, interrupts are disabled during the entire outputting of a line to the printer. However, the time delay will probably be no worse than that associated with disk I/O. If the interrupts are used, printing will nevertheless stop if the foreground program never sends anything to the spooler or tests the keyboard for input. This is because the disk I/O to read the next sector is done only during keyboard checking or main program output to the spooler. See circular buffer discussion for an additional disadvantage when the interrupts are used.

The program asks if the circular buffer is to be used to buffer keyboard input characters. Reply Y if yes; N if no. The circular buffer helps prevent lost keyboard input. If the 25ms interrupt is enabled to drive the printer (see above option), the circular buffer uses the ROM keyboard character input routine and therefore disables any drivers (such as NEWDOS/80's keyboard intercept routine, lower case driver, etc.) activated before the spooler is activated. If the 25ms interrupt is not used to send spooled output to the printer, then that does not frequently check the keyboard or send output to the spooler. The disadvantage of

using the interrupts is that, for a buffered printer, interrupts are disabled during the entire outputting of a line to the printer. However, the time delay will probably be no worse than that associated with disk I/O. If the interrupts are used, printing will nevertheless stop if the foreground program never sends anything to the spooler or tests the keyboard for input. This is because the disk I/O to read the next sector is done only during keyboard checking or main program output to the spooler. See circular buffer discussion for an additional disadvantage when the interrupts are used.

The program asks if the circular buffer is to be used to buffer keyboard input characters. Reply Y if yes; N if no. The circular buffer helps prevent lost keyboard input. If the 25ms interrupt is enabled to drive the printer (see above option), the circular buffer uses the ROM keyboard character input routine and therefore disables any drivers (such as NEWDOS/80's keyboard intercept routine, lower case driver, etc.) activated before the spooler is activated. If the 25ms interrupt is not used to send spooled output to the printer, then the regular keyboard routine(s) (as existed in the 4016H - 4017H vector at spool activation) is used. This latter also holds if the circular buffer is not used, regardless of whether or not the 25ms interrupt is used.

Now that the spooler has all the initialization parameters, the in-main-memory program is altered. The program then asks for the filespec of the working program module to be stored on disk. Respond with the filespec you will use in the filespec2, A DOS command discussed below; do NOT respond ASPOOL/MAS!!!!!! The working program module will be written to disk, and the spool program exits to DOS via 402DH. HINT: Use SPOOLER/CMD for filespec

3. ACTIVATE SPOOLING. When spooling is to be used, enter the DOS command "filespec2,A" (example: SPOOLER,A) where filespec2 is the filespec of one of the working spool program modules you have created. filespec2 must **NEVER** be ASPOOL/MAS. If the spooler is already active, 'FILE ALREADY EXISTS' error message is displayed.

The module will load into the 5200H - 5FFFFH region, relocate itself to HIMEM+areasize+1, and sets HIMEM = HIMEM+areasize where HIMEM is the DOS high memory address contained in Model I locations 4049H - 404AH (Model III locations 4411H - 4412H) and areasize is the amount of memory required by the spooler. Then the keyboard vector at 4016H - 4017H and the printer vector at 4026H - 4027H are intercepted to vector to the spooler. If interrupts are to be used, a routine is entered into NEWDOS/80's 25ms interrupt chain of user interrupt routines. 'SPOOLER ACTIVE' is displayed, and the 402DH exit is taken to DOS.

The spooler is now active. All data intended for the printer will be directed to one of five disk files (POOL1, POOL2, POOL3, POOL4, POOL5). Why five files you may ask? Well, when you have "printed" as much data as you wish and would like that data to be actually printed on the real printer, you send an end-of-file to ASPOOL. This is done either via DOS command *ASP,W (CMD"*ASP,W" from BASIC) or by outputting to the spooler a 03 byte in the normal print stream

(LPRINT CHR\$(3) from BASIC). The file that was spooling will be closed and scheduled for printing. You may now spool to another file by just "printing" more data. The data will be placed on the disk while the first data file is

being printed. This procedure may be repeated five times. If you try to spool a sixth file before the first has been printed on the real printer, the system will display 'SPOOL FULL. WAITING ON PRINTER' and will hang until a file is printed. All data is printed on the real printer in the background while the current or another main task is executing or simply while the system is waiting for the user to tell it what to do next. Whenever *ASP,W is executed or a 03 byte is seen in the output to the spooler, the spooler program considers this an end of file (performing top-of-form if specified) even though you may be sectioning your spooled output for one report to keep the printer going and avoid running out of space.

Warning!!! The Model III ROM routine, normally used by the spooler, will discard the current character being sent to the printer if it senses the printer is not ready (including busy) and the BREAK key is pressed. Since the executing foreground program may be using the BREAK key while the spooler is printing in the background, there will be times when printer characters will be lost, unknown to the spooler. This can seriously limit the usefulness of any spooler on the Model III that uses the ROM printer driver routine.

You may bring the spool system down gracefully at any time by the DOS command *ASP,S (CMD"*ASP,S" from BASIC) or by sending a 04 byte in the normal output to the spooler (LPRINT CHR\$(4) from BASIC). This procedure will purge the current spool file, will prevent any new files from being created, and will display 'SPOOL STOPPING'. Main program execution then continues, any characters sent to the spooler will be ignored and the spooler continues to print any files that have been scheduled. When all files have been printed, the *ASP,P function is performed. NOTE, if the spooler appears to hang, it is probably waiting for the main program to check the keyboard. If the main program can't do this, try DFG, but wait till the drives stop.

You may bring the spool system down abruptly at any time by entering DOS command *ASP,P (CMD"*ASP,P" from BASIC). All remaining spooled data is lost. If an interrupt routine was active, it is purged. The keyboard and printer vectors are restored to what values they were when the spooler activated. If DOS's HIMEM value is the same as that set by the spooler when activated, HIMEM is set back to what it was before the spooler was activated, thus reclaiming the spooler's main memory. However, if the HIMEM is not the same, HIMEM is not changed, and the spooler memory remains lost to subsequent main programs. 'SPOOLER PURGED' is displayed, and the DOS 402UH exit taken to DOS.

You may flush the print queue at any time by entering DOS command *ASP,C (CMD"*ASP,C" from BASIC). The spooler will respond with "CLEAR BACKLOG OR PRINT (B/P)?". Respond with a B and Enter if you wish to clear the backlog, or a P and Enter to stop printing the current print file. Clearing the backlog does not purge the current print file, and clearing the current print file does not purge the backlog.

The status of the spool system may be determined at any time by entering the DOS command *ASP (CMD"*ASP" from BASIC). The system will print a list of all files waiting to be printed (BACKLOG) and any file that is open for printing or spooling. If the system has been stopped but not yet purged, "SPOOL STOPPING" will be displayed. If the spooler has been purged or not activated, 'FILE NOT IN DIRECTORY' is displayed.

7. DISK BASIC, NON-I/O ENHANCEMENTS

7.1. INTRODUCTION, Requirements

For NEWDOS/80 most, but by no means all, of the interface specifications between BASIC and the BASIC programmer remain the same as for DISK BASIC under TRSDOS 2.3 on the Model I and for TRSDOS 1.3 on the Model III. The NEWDOS/80 BASIC user is expected to have and be knowledgeable of both the non disk BASIC manual and the disk BASIC portions of the TRSDOS manual for whichever of the two TRS-80 models is being used. The current and next chapters of this NEWDOS/80 version 2 documentation discuss only the differences from the TRS versions. Both the Tandy manuals are excellent; if they didn't come with your TRS-80 when you bought it, buy them!!!! Apparatus does not, in this manual, duplicate their contents.

7.2. General comments

1. When a BASIC syntax error occurs, BASIC does not automatically enter EDIT on the offending text line, but it does set that line as the current line. If the operator wishes to edit the line, press comma. This change is to make it more difficult for the operator to inadvertently clear variables that he/she would otherwise want to see to assist in debugging.
2. BASIC programs may disable the BREAK key via CMD"BREAK,N", and re-enable it by CMD"BREAK,Y".
3. Because CLOAD does a NEW function between consecutive bytes from tape, it will lose synchronization if BASIC is running with more than 3 file areas.
4. When a DOS error is encountered by BASIC and if no ON-ERROR routine is active, both the DOS error message and the BASIC error message are displayed.
5. BASIC now has a total of 8 overlays that it uses. The user will notice that disk I/O occurs whenever RUN is executed and whenever execution is interrupted (STOP, error or BREAK) or terminated (END); this is done to bring in BASIC routines needed for the current or anticipated next function.
6. NEWDOS/80 DISK BASIC does NOT allow text line deletion to be done by simply typing in the line number. The explicit delete command, DELETE or D, must be used.

7.3. Activating DISK BASIC

DISK BASIC is activated by keying in one of the following commands to DOS:

1. BASIC
2. BASIC
3. BASIC n
4. BASIC m
5. BASIC cmd
6. BASIC n,m,cmd
7. BASIC m,n,cmd
8. BASIC n,m
9. BASIC m,n
10. BASIC n,cmd
11. BASIC m,cmd

where:

***** means the user wants BASIC to reinstitute the program in the text buffer, using the same values for m and n as appear to exist in main memory. This allows the user to recover from an unwanted 'reset' or to get back to the same program after a CMD"S". If BASIC is able to accomplish this, it forces 'LIST' as its first command. If BASIC is unable to reinstitute the program, it exits to DOS READY. BASIC * will not work if n was less than 2 or if the program was less than 3 lines.

n = the number of fileareas that BASIC is to allocate, default = 3, maximum = 15. This is the highest fan (filearea number) that will be used in any statement during this invocation of BASIC. If the BASIC program is to use field item files with standard record length not equal to 256, then n must be specified and must be suffixed with the character V (see example 4 below).

m = memory size. The value m minus 1 is the highest memory location that BASIC is allowed to use. If m is not specified, the current DOS HIMEM value is used. All memory m and above is not used by BASIC and can be used for other routines such as printer drivers, special code USR routines, etc.

cmd = one line of BASIC text, consisting of one or more BASIC statements. This text line is considered direct keyboard input and will be executed as soon as initialization is completed.

Remember, the DOS command activating BASIC is limited by DOS to a maximum of 80 characters, including ENTER, and it is further limited to 32 characters, including ENTER if invoked via 'AUTO'.

Any error encountered during initialization causes a return to DOS.

If DOS is in RUN-ONLY state, the DOS command activating BASIC must contain a RUN or a LOAD (option R) statement.

Examples:

1. BASIC Brings up BASIC with 3 file areas, high memory set to the current value for HIMEM in DOS and displays 'READY', waiting for the operator's command.

2. BASIC,RUN"XXX/BAS" Brings up BASIC with 3 file areas, high memory set to the current DOS HIMEM value, loads BASIC program XXX/BAS into the text area and starts its execution.
3. BASIC,9,48152,LOAD"XXX/BAS" Brings up BASIC with 9 file areas, high memory set to 48151 (1 less than 48152), loads BASIC program XXX/BAS into the text area and displays 'READY', waiting for the operator's command.
4. BASIC,3V This works the same as example 1 above, except that each of the 3 file areas is assigned an extra 256 byte buffer. This extra buffer per filearea is needed if the program will be using field item files with a record length other than 256.
5. BASIC,CLEAR3000:A=1:RUN"XXX",V Brings up BASIC with 3 fileareas, sets its high memory value to DOS's current HIMEM value, performs CLEAR reserving 3000 bytes for the string area, sets numeric variable A equal to 1, loads BASIC program XXX and commences its execution without clearing the variables, thus leaving variable A intact for the program to inspect.

7.4. Direct Scrolling/Editing Commands

NEWDOS/80 DISK BASIC allows the following 'direct' commands:

- . (period) LIST the current text line.
- down-arrow** LIST the next text line. If there is no next text line, performs as / .
- up-arrow** LIST the text line before the current line. If none, performs as ; .
- ;** or **shift-up-arrow** LIST the first text line.
- /** or **shift-down-arrow** LIST the last line in text. Users having the newer ROM will find that shift-down-arrow is no longer a usable key; hence the need for / .
- :** Scroll one display page toward the start of the text. When done, the previous current text line is now at the bottom of the display excepting that if the previous command was . or @ , the previous display's top line is now the new display's bottom line. The new current text line is the bottom line on the new display.
- @** Scroll one display page toward the end of text. When done, the previous current text line is now at the top of the display, and the new current text line is the bottom text line on the new display.
- ,** (comma) EDIT the current text line.

Only 1 such command per direct statement line, and the command, to be seen, must be the first character of the input line (no line number or backspacing allowed).

7.5. Text Editing Command Truncation

NEWDOS/80 DISK BASIC allows the truncation of the commands AUTO, DELETE, EDIT and LIST to A, D, E and L respectively when the following conditions are met:

1. 1st character of the input line.
2. Followed by either a period or a decimal digit.
3. The input line does not contain an =.

7.6. DI and DU text editing functions

DI and DU Two additional BASIC text editing functions are implemented using the following forms of direct command:

1. DI aaaaa,bbbb
2. DI .,bbbb
3. DU aaaaa,bbbb
4. DU .,bbbb

where:

aaaaa is the line number of the text line to be moved or duplicated, and **bbbb** is the line number to be given the moved text line or the duplicate of the text line.

DI means to delete the line at aaaaa and insert it at bbbb.

DU means insert at bbbb a duplicate of the text line at aaaaa, but do not delete the line at aaaaa.

Text referring to aaaaa is not altered to refer to bbbb. If this is desirable, then use RENUM to move the text line.

The use of a period in place of aaaaa causes aaaaa to default to the last line listed, edited or deleted.

7.7. RUN and LOAD (optionally retaining variables)

RUN and LOAD may now optionally retain all variables and open fileareas by using the V option in the following formats:

```
RUN "filespec1",V
LOAD"filespec1",V
```

where filespec1 is the filespec of the program file being executed. The LOAD with the V option executes exactly the same as the RUN with V option. The RUN with V option preserves all the variables, excepting DEFFN variables, during the execution of RUN; thus the variables existing before the RUN statement can be used after the RUN statement. Any fileareas open prior to the RUN are left open for use after the RUN statement. If the V option is specified, the R option may not be. See example 5 in section 7.3.

7.8. MERGE Dynamic loading of overlay program

The MERGE statement has been expanded:

MERGE will merge either an ASCII or a packed text file.

MERGE may be executed as a direct statement or as a program statement.

If MERGE is executed as a program statement, the MERGE statement must not be part of a DEFFN statement, a subroutine or a FOR-NEXT loop (as a POPS function is implicitly performed), must be the last statement of the text line, must be followed by the text line where execution will continue after the MERGE, and the merge file must not contain a line whose number is the same as the number of a text line existing at the start of the execution of the merge (use CMD"F",DELETE to delete conflicting text lines before executing the MERGE). The merge protects all variables. The user must assure enough main memory space is available for the merge as error recovery is not possible if the merge fails once actual merging commences. Example:

```
100 MERGE"XXX/BAS"
110 X=1           execution continues here after the MERGE is completed
```

7.9. RENUM Renumber the Current BASIC Program.

```
RENUM sssss,iiii,ppppp,qqqqq[,U][,X]
RENUM ,
RENUM U
RENUM X
RENUM U,X
```

The current BASIC program or a part of it may be renumbered while it resides in the text area. Via the U option, the RENUM does not actually perform renumber but only does its text error checking, thus allowing the undefined line numbers and some, but not all, syntax errors to be found. The user may, by proper choice of the new line number values, move a portion of the program to a different place in the program with all references to any of the moved lines changed to the new line numbers. Lastly, via the X option, RENUM will not declare as an error any undefined line number if that line number lies outside of the range of lines being renumbered, thus allowing a program to have references within it to lines that are intentionally not part of the program.

The basic renumber command causes all text lines whose line numbers are greater than or equal to ppppp and less than or equal to qqqqq to be assigned new line numbers. sssss is the first new line number assigned with subsequent numbers generated by adding iiii to the line number of the previous text line. sssss and iiii must be in the range 1 - 65529 and have default value 10. ppppp must be in the range 1 - 65529, has default value 0. qqqqq must be in the range 1 - 65529, greater than or equal to sssss, and has default value 65529. The range of newly generated line numbers must not encompass any old text lines that are not part of the resequenced range ppppp - qqqqq inclusive. So long as this rule is observed, the newly generated line number range may be placed anywhere in the text with the renumbered text moved to the proper new text location.

At least one parameter must be specified. If the user wants to specify all defaults and neither X nor U, then use a comma as the only parameter.

For the series sssss,iiii,ppppp,qqqqq, if one or more of the 4 numbers are to use the default values, then commas must appear in the proper place to indicate which of the 4 values a given line number is for. See example 4 below.

If the U option is specified, the text is not altered in any way and RENUM simply searches text for undefined line numbers and for some errors associated with BASIC statements that use line numbers. These errors are displayed in the following format:

```
sssss/U - there is no text line sssss.  
sssss/X - text line sssss has syntax error.  
sssss/S - text line sssss has a bad line number.
```

If the X option is specified, references to non-existent text lines are not displayed as errors if that line number is also outside of the ppppp to qqqqq range. The X option is intended as aid to programmers who use a base program and overlay programs which refer to text lines in each other.

If any error is encountered before text is altered, the command reverts to performing as if the U option had been specified and displays all the errors it can find. If an error is encountered after text alteration begins, 'FATAL ERROR. TEXT NOW BAD' is displayed and the 4030H exit taken to DOS. The BASIC text must not be reclaimed (don't use BASIC *).

If either SYS11/SYS or SYS13/SYS are not in the system when RENUM is executed, the system will exit to DOS READY (see section 5.5).

RENUM will refuse to renumber a program whose first text line's number equals 0. Use 'DI' to assign the line a number other than 0. Examples:

1. RENUM U The BASIC text is checked for undefined line numbers and other errors that would normally be encountered in an actual renumber. The BASIC text is not altered.
2. RENUM , The entire BASIC text is renumbered using an increment of 10. The first text line is assigned line number 10, the 2nd assigned line number 20, and so on.
3. RENUM 100,100 The entire BASIC text is renumbered using an increment of 100. The first text line is assigned line number 100, the 2nd is assigned 200, and so on.
4. RENUM 2050,,2050,3160 All text lines from and including any line numbered 2050 to and including any line numbered 3160 are renumbered using an increment of 10. The first renumbered line is assigned line number 205\$, the second is assigned 2060, and so on.
5. RENUM 30000,5,15365,18112 All text lines from and including any line numbered 15365 to and including any line numbered 18112 are renumbered using an increment of 5. The first renumbered line is assigned line number 30000, the 2nd is assigned 30005, and so on. The renumbered text lines are moved to the new positions in the text.

7.10. REF List references to variables, line numbers and keywords

The BASIC statement REF allows the BASIC programmer to find all places in the program where a line number, an integer, a variable, a string, a function code, a packed sequence of characters or an unpacked sequence of characters is referenced. REF has the following formats:

1. REF* Display full reference list for all line numbers, integers and variables.
2. REF\$ Print on the printer a full reference list for all line numbers, integers and variables.
3. REFnn Display all references to the variable(s) named nn. If nn is only 1 character, a blank is assumed for the second. nn may not be more than 2 chars and must not have a type suffix.
4. REFsssss Display all references to the line number and/or integer sssss where sssss is a 1-5 decimal digit number between 0 and 99999. Hexadecimal or octal references within the text are not listed.
5. REF*nn
6. REF\$nn
7. REF*sssss
8. REF\$sssss
9. REF Display the next text line containing at least one reference to the variable or number specified by the last REFnn or REFsssss statement executed. If there are no more referencing text lines, 'TEXT END' will be displayed. If 'REF' entered again, the first referencing text line will be listed. Remembrance of the search variable name or number and the current search line number within the text is usually (but not always) lost when any command involving DOS is executed.
10. REF=xxx The character sequence xxx is packed by the standard BASIC text packing routine. The BASIC text is then searched for a match on the packed xxx value and the line numbers listed for all lines containing the packed xxx value. If the packed value xxx is more than 16 bytes long, only the first 16 packed bytes participate in the compare. This format of REF is to be used when the user wants to know where in the text a BASIC function code (i.e., PRINT, LPRINT, GOTO, etc) is used. The text lines containing xxx can be displayed one at a time by repeated issuance of the format 9 REF command.
11. REF"xxx This format operates similar to format 10 except that xxx is not packed. xxx is considered a string unless xxx itself contains a ". This format allows xxx to be found in strings and comments.
12. REF@sssss This statement is similar to format 9 except that the search will start with 1st text line whose line number is greater than or equal to sssss.

Press BREAK to pause, ENTER to continue, and up-arrow to terminate the REF function. Formats 5-8 are the same as 1 and 2, except listing/printing starts

with the specified variable name or decimal number, if it exists, or the next higher existing name or number, if not.

If SYS12/SYS is not in the system when the REF statement is executed, the system will exit to DOS (see section 5.5).

7.11. Lower Case Suppression (Model I only)

Text String Lower Case Suppression (Model I only) Users who do not have the hardware lower case modification or those that do but don't use a lower case driver to bypass the ROM display routine will occasionally be puzzled why some string compares fail and syntax errors appear in perfect appearing statements. This is due to the acceptance of lower case letters into strings which display as upper, and the acceptance of lower case @ into text statements. Remember the ROM swaps lower case to upper and vice versa before BASIC sees the characters. In the case of data, there is nothing that can be done about this problem except to remember that if it appears equal on the display, there still may be a lower case/upper case mismatch in memory. For text input, if system option AS = Y, text string lower case letters, but not lower case @, will be forced to upper case, eliminating many of these problems.

7.12. RUN-ONLY

For DISK BASIC there are two ways BASIC can be forced to run in RUN-ONLY mode: (1) if system option AB = Y, and (2) if the BASIC program file is password protected, passwords are enabled, the access password specified in the RUN or LOAD (option R) statement and the access level = EXEC.

If system option AB = Y, the DOS command activating BASIC must contain the necessary RUN or LOAD (option R) statement to start a program executing as the operator is not allowed to input any direct command statements.

In RUN-ONLY, the BREAK key is disabled and BASIC is inhibited from accepting direct statements (data is OK) from the operator. The program has full control, and must exercise it. A menu program can issue RUN or LOAD (option R) statements for other BASIC programs, and those programs can do the same to return to the MENU program or go on to the next program of a sequence. Optionally, a base program may remain in memory at all times, and via CMD"F", DELETE and MERGE, bring in overlay programs as necessary. Programmers should carefully study available options under RUN, MERGE, LOAD, and CMD"F" functions.

7.13. Comparisons in the use of the function CMD between NEWDOS/80 and TRSDOS.

1. **CMD"A"** Not implemented; use CMD"S".
2. **CMD"B"** Not used on the Model I by NEWDOS/80 nor TRSDOS. TRSDOS' Model III use is not implemented in NEWDOS/80; use CMD"BREAK,Y/N"
3. **CMD"C"** This command (1) compresses out all spaces from the program text, excepting for those within strings, and (2) deletes all remarks

from the text, including entirely those lines which are entirely remarks. The statement `CMD"C",S` compresses out all spaces from the program text, excepting those within strings and remarks. The statement `CMD"C",R` deletes all remarks from the text, including deleting entirely those lines which were entirely remarks.

In some cases, `GOTO`, `GOSUB`, etc. refer to a text line that is entirely remarks and the deletion of remarks from the text will cause these referenced lines to disappear. The programs should be altered to have these `GOTO`s and `GOSUB`s refer to text lines that are not entirely remarks. After remarks have been deleted from a program, execute `RENUM U` to determine if there are any undefined line numbers resulting.

Though BASIC is designed to ignore spaces that are not in text remarks or character strings, the removal of spaces from text can still cause confusing situations. For example, compressing

```
10 FIELD 1,20 AS C$
20 IF F OR D THEN 10
to
10 FIELD1,20ASC$
20 IFFORDTHEN10
```

will cause syntax errors to occur for both lines during execution after either (1) the program has been stored in ASCII and then read back in or (2) the lines have been edited. To avoid these problems that may exist for weeks or months before either of the above two conditions occur, the `CMD"C` function automatically unpacks each compressed text line, packs it again and compares the new packing with the old that existed before the spaces were compressed out. For any text line where the two packings are different in any way, the spaces are restored into that text line (remarks, if deleted, remain deleted) and the line's number is listed on the display. The user may then inspect these lines and remove spaces that won't affect the program. For any given program, there should be very few lines rejected by `CMD"C`.

4. **`CMD"D`** `TRSDOS`' meaning is not implemented on the Model III under `NEWDOS/80`; use `CMD"doscnd`". On the Model I, `CMD"D` still invokes `DEBUG` though 123 is the preferable method.

5. **`CMD"E`** Displays the DOS error message associated with the latest DOS error encountered by BASIC.

6. **`CMD"F`** Not used in `TRSDOS`. In `NEWDOS/80`, there are two formats:

1. `CMD"F",fc` used when the function code `fc` must be findable by `REF`, `RENUM` and others.

2. `CMD"F=fc` used when the function code `fc` is not to be seen by `REF`, `RENUM`, etc. or where the specially defined function code could be confused by the normal text packing routine.

These `CMD"F` functions are specified in sections 7.15. thru 7.20.

7. **CMD"I** Not used on the Model I by either NEWDOS/80 or TRSDOS. TRSDOS' Model III use is not implemented in NEWDOS/80; use CMD"dos-cmd".

8. **CMD"J** Calendar Date Conversion.

CMD"J",date1,date2

converts the expression date1 to the appropriate format and stores the result in the string variable date2. If date1 is in mm/dd/yy format, date2 is stored in ddd format and if date1 is in -yy/ddd format, date2 is stored in mm/dd/yy format where:

mm is a two digit month value between 01 and 12.
dd is a two digit day-of-the-month value between 01 and 31.
ddd is a three digit day-of-the-year value between 001 and 366.
yy is a two digit relative year-within-century value between 00 and 99. For leap year conversions, yy is assumed to be in the 20th century, i.e., from 1900 to 1999.

9. **CMD"L** TRSDOS Model III meaning not implemented in NEWDOS/80; use CMD"LOAD,filespec". This function is not used on the Model I.

10. **CMD"O** Array Sort; see discussion below (section 7.21.) for CMD"O".

11. **CMD"P** Not used on the Model I. TRSDOS' Model III meaning is not implemented in NEWDOS/80; use PEEK(&H37E8) to obtain the 0 - 255 value for the current printer status.

12. **CMD"R** TRSDOS' Model III meaning is not implemented in NEWDOS/80; use CMD"CLOCK,Y". On the Model I, CMD"R" still reenables the interrupts as before.

13. **CMD"S** Exit BASIC and return to DOS READY state. However, if the command is of the form CMD"S=doscmd", then the following occur:

1. The DOS command doscmd is moved into the DOS command buffer.
2. BASIC exited.
3. The DOS command placed into the DOS buffer is executed immediately without an intervening DOS READY.
4. When that command is completed, control returns to DOS READY and not to BASIC.

14. **CMD"T** TRSDOS' Model III meaning is not implemented in NEWDOS/80; use CMD"CLOCK,N". On the Model I, CMD"T" still disables the interrupts as before.

15. **CMD"X** Not used on the Model I by NEWDOS/80. TRSDOS' Model III meaning is not implemented; use the REF command.

16. **CMD"Z** Not used on the Model I by NEWDOS/80. TRSDOS' Model III meaning is not implemented; use CMD"ROUTE,...".

7.14. CMD"doscnd"

If the string expression associated with the CMD function has two or more characters and does not start with either "S=" or "F=", then the string is assumed to be a command to be executed by DOS. BASIC moves the command to DOS' command buffer, sets DOS to MINI-DOS mode, and calls DOS to execute the command via 4419H, DOS-CALL. Upon return, BASIC turns off DOS' MINI-DOS mode. If DOS has rejected the command because it was not legal under MINI-DOS, BASIC then attempts to reissue the command to DOS under normal mode by doing the following:

If approximately 8,000 bytes are not available between the top of BASIC's array areas and the bottom of BASIC's stack (which is immediately below the string area), BASIC declares OM ('OUT OF MEMORY') error and terminates the current statement. If the space is available, BASIC moves all of memory from 5200H to 70FFH to that free area, sets itself to use stack area 7000H-71FFH and computes a checksum over the region from 7100H to the top of BASIC's memory (takes about 2 seconds). Then it calls DOS to execute the DOS command. Upon return from DOS, BASIC moves the saved region back to 5200H-70FFH and recomputes the checksum (again, another 2 seconds). If the check fails, this means that the DOS command executed has altered some of BASIC's bytes; BASIC cannot continue and exits to DOS with 'BAD MEMORY' error.

Whichever way the command was executed, BASIC now checks the return code from DOS. If an error occurred and the error message has already been displayed, BASIC terminates the CMD"doscnd" statement with 'PREVIOUSLY DISPLAYED ERROR' error state. If a DOS error occurred, BASIC calls 4409H to display the DOS error message and terminates the CMD"doscnd" statement with 'DOS ERROR' error state. If no error occurred, BASIC continues with normal processing.

Any DOS library command or assembly language program (that will execute using only the 5200H - 6FFFH region and/or a non-BASIC, non-DOS region of main memory) can be executed in this fashion. SUPERZAP and DIRCHECK are two programs that may be executed through CMD"doscnd". FORMAT and most forms of COPY can be done; however, single drive, two diskette copies cannot be done as they require the maximum amount of memory. Also, don't specify the UBB parameter in COPY.

Remember, DOS commands are limited to 80 characters, including the ENTER character that BASIC will append to the doscnd string when moved to the DOS command buffer.

User programs are warned to leave the Model I memory area 4080H -41FFH (Model III area 4080H - 41E2H) alone except where alteration is in conformance with BASIC's current uses.

CMD"BASIC" should never be executed. If for some reason the programmer wants to exit BASIC and return, use CMD"S=BASIC".

Almost all DOS commands may be executed via CMD"doscnd". Examples:

1. CMD"DIR 1" list a directory
2. CMD"COPY XXX:0 YYY:1" copy a file
3. CMD"COPY 0 1 07/10/81 FMT" full diskette copy, with format
4. CMD"SUPERZAP" executes program SUPERZAP and return to BASIC
5. CMD"DO CHAINFIL" perform chain file functions and return

7.15. **CMD"F=POPS", CMD"POPR" and CMD"F=POPN"**

If the statement is **CMD"F=POPS"**, then all returns and FOR-next controls are purged, leaving BASIC with no outstanding returns or nexts. When done, execution continues with the next statement. The purpose of this statement is to allow the programmer to 'bail-out' of complex coding and return to BASIC's first level. This avoids leaving residual information in BASIC's control stack which on recursive returns to the high level without **CMD"F=POPS"** will eventually cause program failure.

If the statement was **CMD"F=POPR"**, then the current GOSUB level is purged along with any outstanding FOR-NEXTs for that level. This is the same as return except control does not pass to the statement following the associated GOSUB, but instead passes to the statement following the **CMD"F=POPR"** statement.

If the statement is **CMD"F=POPN"**, then the most recently established FOR-NEXT's control data is purged. This is the same as 'NEXT' where the loop limit is exceeded. Execution continues with the statement following the **CMD"F=POPN"** statement.

If the statement is **CMD"F=POPN" vn** where vn is a variable name, the FOR-NEXT loop associated with vn is purged along with any other FOR-NEXT loops established while vn's loop was outstanding. Execution is the same as for 'NEXT vn' when the loop is to end. Execution continues with the statement following the **CMD"F=POPN" vn** statement. The purpose of **CMD"F=POPN"** is to allow breaking out of a loop while not leaving residual loop control information that can confuse the programmer if he/she subsequently uses FOR-NEXT variables in reverse order.

7.16. **CMD"F=SASZ"**

Change BASIC's string area size without affecting or clearing the variables.

```
CMD"F=SASZ",expl
```

allows the string area size to be changed without clearing the variables. expl must be a value large enough allow the string area to contain the strings that it contains when the statement is executed. An error will be generated if expl is too small or is too large (i.e., will cause overlap with the text, scalar and array areas). Example:

```
CMD"F=SASZ",4000
```

7.17. **CMD"F=ERASE" and CMD"F=KEEP"**

Selective clearing of BASIC variables.

CMD"F=ERASE",vn1,vn2,vn3... allows the specified variables to be cleared. If a specified variable is within an array, the entire array is cleared. The size of the string area is not changed. This statement should be used when an array is no longer needed or the user wishes to redimension it by a subsequent DIM statement. This statement may be multi-text lines as described for **CMD"F=KEEP"** below.

CMD"F=KEEP",vn1,vn2,vn3... causes all variables to be cleared except those specified and except specially defined variables such as those defined by a DEFFN statement. The size of the string area is not changed. If no variable names are specified, all variables are cleared, except the special ones. If a specified variable name is within an array, the entire array is exempted from the clear. The statement may specify as many variable names as desired with overflow from one text line to the next non-comment text line taking place whenever the last variable name of a text line is followed by a comma. Example:

```
CMD"F=KEEP",A$,B%,C,D#,      'statement first line
E!,F,G$,                    'statement 2nd line
REM this line is bypassed
H!,I                        'statement last line
```

7.18. CMD"F",DELETE

Dynamic deletion of text lines:

```
CMD"F",DELETE ln1-ln2
```

This statement allows the text lines from and including any line numbered IS to and including any line numbered ln2 to be deleted during program execution. All variables are retained, excepting that DEFFN variables for DEFFN statements in the delete range are cleared. The string area size is not changed. Any string variable whose current string was actually in the deleted text area has that string moved to the string area. CMD"F",DELETE must not be executed as a direct statement, must not be contained in a DEFFN statement, a subroutine or a FOR-NEXT loop has a POPS function is implicitly performed), must be the last statement on its text line and must be followed by the text line where execution will continue after the delete. Example:

```
100 CMD"F",DELETE 10500-15000
110 X=1      execution continues here after the DELETE is completed
```

7.19. CMD"F=SWAP"

Swapping of variable contents:

```
CMD"F=SWAP",vn1,vn2
```

This function swaps the value of variable vn1 with that of variable vn2. Both variables must be of the same type, i.e., both strings, both single precision floating point, etc. Example:

```
CMD"F=SWAP",A$,B$
```

7.20. CMD"F=SS"

BASIC single stepping:

1. CMD"F=SS" turn on single stepping
2. CMD"F=SS",ln1 single stepping starts at line ln1.
3. CMD"F=SS",N turn off single stepping.

The BASIC programmer may now single step through program execution. Using either format 1 or 2 above sets BASIC into single step mode, though for format 2, actual single stepping does not start until text line ln1 is the next line to be executed. A single BASIC text line is executed for each step, and between steps the line number for the next line to be executed is displayed in '@nnnnn' format in the display upper right corner to indicate that BASIC is waiting for the operator to respond. Responding ENTER causes line nnnnn to be executed and then BASIC waits for user response again. Responding BREAK causes execution to be broken in the normal manner though it should be noted that the line number the BREAK shows is for the line just executed or being executed while the '@nnnnn' display is for the next line to be executed. If the user does not change text during BREAK, the program may be continued via CONT; in this case, the '@nnnnn' display will immediately reappear without execution of a line. Pressing ENTER will then execute the line. While in BREAK, the operator may turn single stepping on or off as desired without affecting the ability to CONT. If the BREAK occurs before RUN or LOAD,R executes one text line, CONT will not work.

Single stepping or the scheduling of the single stepping to start when a particular text line is encountered remains in effect until either CMD"F=SS",N is executed to turn it off or until a format 2 type stepping command is executed, wherein stepping goes off until the specified line is encountered. The execution of RUN, LOAD, NEW, etc. does affect single stepping state.

7.21. CMD"O"

The main memory BASIC array sort has 2 formats:

1. 1. CMD"O",n,av1[,av2,...,1 (direct sort)
2. 2. CMD"O",n,*iav1,av2[,av3,...] (indirect sort)

In explaining this sort, the term REN is used and is defined to mean a Relative Element Number identifying an array element. The elements within any BASIC array, regardless of dimension, are integer numbered from 0 up. If an array has only one dimension, then an element's REN is simply the value of its subscript and if you use only single dimensioned arrays, you can ignore the rest of this paragraph. However, if you use multi-dimensional arrays, then you should know which method to use to increment array subscript values in order to extract elements in the sorted order. CMD"O" does not care what dimension the arrays have; it simply counts off the array elements in the order BASIC stores them in main memory. You, the programmer, do care as you must use subscripts in order to access the array elements. For multi-dimensioned arrays, the rule for computing the REN is complex and can best be illustrated by a three dimension array example using two statements:

```
DIM A(R1,R2,R3)
Y = A(X1,X2,X3)
```

where the REN of this element is computed as $X1+X2*(R1+1)+X3*(R1+1)*(R2+1)$. If the array had only two dimensions, then the REN would be $X1+X2*(R1+1)$, and, of course, if the array had only one dimension, the REN would simply be $X1$.

If the CMD"O" statement specifies more than one array, excluding iavl, then the RENs for the first sort item in each array, excluding iavl, must be equal.

The sorting order used has one level for each array specified, excluding the iavl array, with highest to lowest level in the order, left to right, of the array variables in the CMD statement. Within each level, the normal sort order is ascending ASCII (actually hexadecimal) numeric value for character string arrays and most negative to most positive value for numeric arrays. However, if the array variable in the CMD statement is prefixed with a minus sign (example: -A#(0)), then the order of sort within that level is descending ASCII (actually hexadecimal) numeric value for character string arrays and most positive to most negative value for numeric arrays. A null compare string character is considered to have a numeric value less than

Normally in character compares, the entire string is used in the compare. However, if the array variable in the CMD statement is suffixed with a field of the form (x,y) (Example: A\$(1)(5,4)), then the compare starts with the xth character of the string and compares using only y characters.

n is the number of elements in each of the arrays participating in the sort. Only n elements from each array participate in the sort. Elements of an array below or above the n elements specified do not participate. If n is a zero value, then for the sort, n is set to the number of elements in first array specified from and including the element specified through and including the last element of the array.

If the number of elements in any array from and including the specified element to and including the array's last element is less than n, FC error is declared.

A maximum of 9 arrays may be specified. All array variable subscripts, except for the indirect array if specified, must evaluate to the same REN value.

Format 1 is a direct sort meaning that the elements of all 1 to 9 arrays are moved around to conform to the desired sort order.

av1 must be specified; av2 and up are optional.

The resulting order of the n elements in each array is the same for each array (i.e., the arrays are not sorted independently). Thus, if the jth element of array 1 is sorted into the kth element slot, then for each of the other arrays, if any, the jth element is also placed into the kth element slot.

Format 1 is compatible with TRSDOS Model III BASIC CMD"O" if and only if only one array variable is specified, it is for a string array and n is an integer variable.

Format 2 is an indirect sort. In this sort, only the n elements of array iavl are altered; the other arrays are not changed in any way. The intent of

format 2 is to allow a sorted sequence to be determined without actually changing the arrays supplying the sort values. A user may have a group of data records spread across a number of arrays such that a record consists of one element from each array, with the REN of each of those elements making up the record equaling the record number. By using format 2 with the indirect array, the user may effectively sort the records using a subset of the items as the sort criteria and without actually rearranging the order of the records, thus leaving them in record number order.

Format 2, as opposed to format 1, is indicated by specifying the iav1 array variable, prefixed by an * .

iav1 must be an integer array variable.

av2 must be specified; av3 and up are optional.

The n consecutive elements starting at iav1 are initialized with the RENs corresponding to the n consecutive elements of array av2 (which also correspond to the RENs for the other arrays, if any).

During sorting only array iav1 is altered; , arrays av2 and up are not altered.

Upon completion, the n elements of array iav1 are in the desired sorted order such that by using successive values out of array iav1 as subscripts, the user may access elements from any of the other arrays (that are single dimensioned) in that sorted order. Accessing multi-dimensioned arrays is more complex and is left as an exercise for the more advanced user.

Example program using a number of sorts:

```
10 DIM NM$(200),AM!(100),LN$(100),IX%(100),ZC!(50),L$(50)
30 X=150
40 CMD"O",X,NM$(0)
60 CMD"O",X,-NM$(25)
70 CMD"O",0,-AM!(1),LN$(1)(5,3)
80 CMD"O",100,*IX%(0),ZC!(1),L$(1)
```

At line 40 the first 150 elements of array NM\$ (elements NM\$(0) to NM\$(149)) are sorted in ascending order. If any of the strings are null, they will appear first in the resulting array. The last 51 elements of array NM\$ (elements NM\$(150) to NM\$(200)) do not participate in the sort and are left unchanged.

At line 60 elements NM\$(25) through NM\$(174) are sorted into descending order, with null strings, if any, appearing as the end elements of those 150 elements. The first 25 and the last 26 elements of the array do not participate in the sort.

At line 70 the AM! and LN\$ arrays are both sorted, both in the same order which is first by descending order of AM! array values and then, where AM! array values are equal, by ascending order of LN\$ array values where only the 5th, 6th and 7th characters of the LN\$ array elements participate in the sort determination. If a LN\$ array element has less than 5 characters, it is considered a null for sort determination purposes. AM!(0) and LN\$(0) do not

participate in the sort. Since the number of elements to be sorted was specified as 0, the number of elements to be sorted was taken as 100, the number of elements in the AM! array from and including the AM(1) element to and including the last element of the array.

Line 80 contains an indirect sort. In this sort, the first 100 IX% array elements are initialized sequentially with REM numbers from 1 to 100 with IX%(a) = 1 and IX%(99) = 100. These RENs are used as subscripts to index into the ZC! and L\$ arrays. The sort is in ascending order, first by ZC! array values and then, where the ZC! array values are equal, by L\$ array values. None of the elements of the LC! and L\$ arrays are changed in any way. Instead of moving the ZC! and L\$ array elements, only the corresponding REM in the IX% array is moved. Upon completion of the sort, the REN in IX%(0) can be used as a subscript to index the first-in-sorted-order element from each the ZC! and L\$ arrays, and the REN in IX%(99) can be used to index the last-in-sorted-order element from each the ZC! and L\$ arrays. Lastly, remember that elements IX%(100), ZC!(0) and L\$(0) did not participate in the sort in any way.

7.22. RENEW

Reinstate a program deleted by the command NEW.

RENEW

The BASIC direct command RENEW reinstates the BASIC program text ostensibly deleted by a just given NEW command. All that RENEW does is set the first byte of the text area non-zero, reestablishes the text forward queue pointers and performs CLEAR. The previous program should thus be reinstated in the text area, available for editing and executing. However, if at least one text line was created or loaded since NEW, then the previous text is not reinstated. Furthermore, if, during this BASIC invocation, the text area never contained any text, RENEW will never the less assume that there is text in the text area and attempt to reinstate it with very disastrous affects to BASIC.

8. BASIC DISK I/O ENHANCEMENTS AND DIFFERENCES

8.1. Introduction

This chapter deals with the substantial enhancements and some differences in the NEWDOS/80's BASIC's file handling over that offered by NEWDOS/21, TRSDOS 2.3 for the Model I and TRSDOS 1.3 for the Model III. The statements made in section 7.1 apply to this chapter as well.

These I/O enhancements are more difficult to understand than they are to use, something like electricity which few understand and everybody uses. In the long run, the enhancements will make I/O programming easier, but the user must remember that since TRSDOS does not have these enhancements, your programs will no longer run on TRSDOS.

In NEWDOS/80 version 1, appendix A of the documentation and an executable, heavily documented BASIC program named SAMPLE01/BAS were included as examples and non-specification discussions of these I/O enhancements. In version 2, SAMPLE01/BAS has been dropped from the diskette and Appendix B added containing 18 example programs on marked and fixed item file usage.

Chapter 8 is intended as the specifications for these enhancements; appendices A and B contain supplementary discussion and examples. If there is a conflict between chapter 8 and appendices A and B, chapter 8 governs.

Many terms used in this chapter are defined in the glossary in chapter 10, which the user will need to refer to. The reader should read through this chapter and appendices A and B at least twice before bogging down trying to understand any particular statement.

8.2. File Type

To the previously existing DISK BASIC file types, sequential which will be called print/input, and random which will be called field item, two other file types have been added: marked item, which has three subtypes MI, MU and MF, and fixed item, which has two subtypes FI and FF.

Print/input (sequential) disk files and field item (random) disk files are well specified for the Model I in the TRSDOS manual, chapter 7 and for the Model III in the TRSDOS manual, part III. The user is expected to have studied the appropriate section before proceeding further with this chapter of the NEWDOS/80 documentation. If necessary, run some test programs to gain proficiency.

A field item file (known in TRSDOS as a random file) has all of its records the same length. This length may be from 1 to 256 bytes. If the record length is other than 256, the BASIC initialization sequence (see section 7.3) must specify the number of fileareas to be allocated and that number must be suffixed with the character V. Example:

```
BASIC,3V
```

will cause BASIC to allocate three fileareas with two buffers each, the first to be used in conjunction with the FIELD statement and the second to serve as a full sector buffer. Remember, this special V suffix is to

be used only if the intention is to use a field item file (TRSDOS random) with a record length less than 256; otherwise the extra 256 bytes allocated to each filearea is wasted. The open statement used where the record length is less than 255 is:

```
OPEN "R",fan,filespec1,lrec1
```

where lrec1 is the logical record length and has a value 1 - 256.

8.3. File type differences

The essential differences between the four file types are as follows:

Print/input files can only be used sequentially; field item, fixed item and marked item files can all be used either sequentially or randomly.

Print/input files are stored in all ASCII character format, converting all numeric data from binary bits to decimal characters. Field item, fixed item and marked item files all store numeric data in the binary forms, thus usually saving disk space and data conversion time.

Print/input files are written to using the PRINT statement which is cumbersome to use because of the need to use the 5 character sequence ";"; to separate two string items. Field item, fixed item and marked item files are written to using the PUT statement with implied separation of file items taken care of by the FIELD statement for field item files, by the implicit or explicit item lengths specified in the IGETL for fixed item files and by the item marker for marked item files.

Print/input files are read using the INPUT statement while field item, fixed item and marked item files use the GET statement.

Field item files require that data be moved into the record buffer before execution of the PUT statement. This is done via the RSET or LSET function and in the case of numeric values, also with MKD\$, MKI\$ or MKS\$ function. This explicit conversion is not needed for print/input, fixed item and marked item files.

Field item files require that numeric data input from the file be converted from string representation to numeric via the CVD, CVI or CVS function before it is used. This is not needed for print/input, fixed item and marked item files.

Print/input files allow a record length of any size. Field item files allow a maximum record length of 256. Fixed item and marked item allow a maximum record length of 4095 bytes.

Print/input file processing transmits strings to the file without change, but truncates leading spaces from string items when inputted from the file. Strings in field item files are padded on either the left or the right with spaces as necessary during the associated LSET or RSET. Strings in fixed item files are padded on the right with spaces as necessary to fill out the item to its specified length or are truncated on the right if the actual string length exceeds the length allowed the file

item. Strings in marked item files are not padded, though the string may be truncated on the right if it exceeds the maximum characters allowed for that item. Except for this truncation, which must be specified by the programmer, marked item file processing is the only one of the 4 that transmits strings completely unchanged from what they were in the corresponding BASIC variable.

8.4. Components of GET and PUT

GET and PUT statements execute in two distinct phases in the following order:

1. File positioning phase. The position within the file is set according to the file position parameter, the second parameter, of the GET or PUT statement.
2. Data transfer phase. The data is transferred from main memory to the file (PUT statement) or from the file to main memory (GET statement).

Before proceeding, it is necessary to define three terms used within GET and PUT statements, one that existed in a more limited form in field item file GET and PUT statements and two that are new.

8.4.1. fp File position. For each GET or PUT operation (see sections 8.8 and 8.9), the file is initially positioned according to the fp specification. fp is one of the following forms:

8.4.1.1. null If REMRA is valid and file record segmented, the filearea is advanced to the next record; otherwise fp = null performs as fp = *. Example:

```
PUT 1,,1000
```

8.4.1.2. * The filearea position is unchanged. fp = * cannot be used to advance from one record to the next for a record segmented file. Example:

```
GET 1,*,1000
```

8.4.1.3. # The filearea is repositioned to REMRA (see section 8.10). This allows the previously processed record to be processed again. Error if REMRA currently invalid. Example:

```
PUT 1,#,1000
```

8.4.1.4. \$ The filearea is repositioned to REMBA (see section 8.10). This allows a return to the positioning of the previous GET/PUT with fp = null, *, #, \$, rn, or !rba. Error if REMBA currently invalid. Example:

```
GET 1,$,1000
```

8.4.1.5. Z See section 8.11 for pseudo FIELD statement discussion.

8.4.1.6. & See section 8.9.6 for PUT, fan,&' discussion.

8.4.1.7. && See section 8.9.7 for PUT fan,&&

8.4.1.8. !rba rba is an expression evaluating to a RBA equalling the desired relative byte position within the file, range to 16,777,215. GET or PUT data transfer starts at the specified location in the file. If the file is record segmented, !rba is assumed to specify a record start position. Example:

```
GET 1,11357,1000
```

***** Use of !rba is extremely powerful and when improperly used, quite disastrous!!!!!!!

***** the expression for fp cannot contain a function, such as LOC, that refers to a filearea.

8.4.1.9. !% Same concept as !rba except the current EOF value is used as the RBA. Example:

```
GET 1,1%,1000
```

8.4.1.10. !\$rba Position the file to relative file location rba. No data transfer is done. See GET discussion, section 8.8.6. Example:

```
GET 1,1$1354
```

8.4.1.11. !\$% Same concept as !\$rba except the current file EOF value is used as the RBA. Example:

```
GET 1,1$%
```

8.4.1.12. !#rba Set the expression rba as the new EOF value. See PUT discussion, section 8.9.9. Example:

```
PUT 1,1#1354
```

8.4.1.13. rn An expression that evaluates to an integer in the range 1 - 32767 representing the target record's number within the file. The filearea is positioned to the start of the record's first item. The filearea must be open with m = I, R or D and with ft, if specified, = FF or MF. Example:

```
GET 1,30
```

8.4.2. IGEL Item Group Expression List. A list of expressions corresponding to a group of file items. An IGEL is a series, terminated by a semicolon, of one or more expressions, separated by commas, corresponding to successive file items, starting at the current file position which was established by the GET or PUTS file positioning parameter. If, while searching for a separating comma, the terminating semicolon or the start of

an expression, a remark or EOL is encountered, the search goes on to the next BASIC statement. The purpose of an IGEL is to serve as the link between a group of file items and a group of BASIC variables or expressions during the execution of a GET or PUT statement for marked or fixed item file processing. Examples of IGELs (coded in BASIC) are:

1. (30)LN\$, (15)FN\$, AM!, DT#(X);
2. "3", AN%, NM\$;
3. (32)A\$(X,Y), B%(2+X), C!, E\$, '1st line
 K#,FS\$; '2nd line

If an error is encountered while processing an IGEL, the error line number will refer to the line containing the associated GET/PUT statement rather than the actual error line within the IGEL.

8.4.3. IGEL expression One of the expressions of an IGEL. For PUT statements, an IGEL expression specifies the value to be assigned to the current file item. For GET statements, an IGEL expression specifies the variable to receive as its value the value of the current file item. An IGEL expression is of one of the following forms:

1. exp
2. (len)exp
3. (len)\$ fixed item files only
4. (len)#
5. a null expression

where:

8.4.3.1. exp is the main portion of the IGEL expression. Normally, exp names a BASIC variable, but in the case of PUT to a marked item file, exp can be almost anything legal on the right side of a LET statement. When exp is a named variable, either a scalar or an array, it is STRONGLY recommended, though not required, that the variable name be suffixed with one of the 4 type symbols (\$, X, I, or F) for example, we STRONGLY recommend:

```

A$,BX(X,Y),C1,D#;
instead of
A,B(X,Y),C,D;
```

This recommendation does not apply to subscript variables (i.e., X and Y in the above example).

8.4.3.2. (len)exp is a prefixed expression with len itself an expression evaluating to an integer 0-255. (len)exp must be used only for IGEL expressions that are strings.

1. For marked item files, len is the maximum number of string characters sent to the file during PUT or received from the file during GET. If the actual number of characters is less, then only the lesser number of characters is transferred. For marked item files, use of the (len)exp format instead of the exp format for string expressions is optional, though for MF files, use of the (len)exp is recommended.

2. For fixed item files, the (len)exp format must be used for string expressions in the IGEL as len specifies the exact number of characters a string file item has or is to have. During PUT statement data transfer, if a variable's string has less than len characters, the file item (not the variable) is padded on the right with spaces as necessary. If the variable's string has more than len characters, the excess characters on the right are not transferred to the file item. During GET statement data transfer, a variable's string receives len characters from the file.

3. Example of IGEL using (len)exp expressions:

```
(30)LN$, (20)FN$, AN%, DP#, (2)CD$(X);
```

8.4.3.3. (len)\$ This expression is legal for fixed item files only. len indicates the number of file bytes to be bypassed. For a GET the specified number of file bytes are bypassed. For a PUT on an existing record, the specified number of file bytes are bypassed and are not altered. For a PUT for a new record, (len)\$ defaults to (len)#. Example, in the following IGEL, the 1st 10 bytes are skipped, the next 12 transmitted, the next 17 are skipped, and the last 8 are transferred.

```
(10)$, AN%, (10)ST$, (17)$, DP#;
```

8.4.3.4. (len)# For fixed item files, for a GET, (len) operates the same as (len)\$ and for a PUT sends len zero bytes to the file. For marked item files, for a GET, (len) bypasses the current file item and for a PUT, sends to the file a character string of len nulls (hex 00 characters). Example:

```
(10)#, AN%, (10)ST$, (17)#, DP#;
```

8.4.3.5. A null expression A null expression can only be used in marked item file GET statement IGELs. A null expression causes bypassing of the corresponding file item. For example, the first, second and fourth items are bypassed in the execution of the statement:

```
GET 1,,,,,X!,,A$;
```

During the processing of an IGEL, if an error occurs particular to one of the expressions of the IGEL, the error message will be prefixed with the expression's position within the IGEL. For example, if the 4th IGEL expression is in error, the error message will be prefixed with a 4.

8.5. Fixed item file characteristics

1. Contains zero or more items.
2. The type and length of each item is determined by the GET's or PUT's associated IGEL, and is not determinable from the file itself. This is a basic difference between fixed item files and marked item files.
3. A file may be subdivided into records all of the same length.
4. Maximum length of records is 4095 bytes.
5. The number and characteristics of items of a record is dependent solely upon record length and the IGEL(s) used to GET or PUT the record.
6. An I/O link to and/or from a fixed item file is created by BASIC statement OPEN with ft = FI or FF.
7. Via the GET statement, the contents of fixed item file items are moved into the BASIC variables specified by the IGEL.
8. Via the PUT statement, fixed item file items are created or replaced from the BASIC variables specified in the IGEL.
9. BASIC statement CLOSE terminates an I/O link between the program and a fixed item file.
10. No disk space is skipped between successive items of a file or between the end of one record and the beginning of the next.
11. When an FF file record is created, any unused space at the end of the record is filled with zero bytes.

8.6. Marked item file characteristics

1. Contains zero or more items.
2. A marked item file item always starts with a control (or marker) byte followed by zero or more additional control bytes followed by zero or more data bytes.
3. Marked file items have the following formats, depending upon the hexadecimal value of the 1st control (or marker) byte.
 1. 80-FF 0-127 byte binary string follows.
 2. 70 SOR (start-of-record). Each record of a MU file (marked item file segmented into records not all of the same length) starts with this item.
 3. 00 Fill item. Used as necessary to fill out MF or MU file records.

4. 71 Next byte contains the count (0-255) of binary string bytes following. This is the only situation (for now) where a second marker byte is used.

5. 72 Next two bytes are a two's complement binary integer. This is BASIC's format.

6. 73 Next four bytes are a binary floating point number in BASIC's format of the form:

1. Three bytes of normalized absolute value mantissa of the form .mmmmm where mmmmm is expressed in these bytes in ascending order of magnitude:

1. Inter-byte, left to right.

2. Intro-byte, right to left. Excepting that the highest ordered mantissa's bit's position, since it's mantissa value is always = 1, is used instead to contain the mantissa sign, 0 = + and 1 = -.

2. The 4th byte contains the base two exponent, biased 128, except if the byte = 0, then the floating point number = regardless of the contents of the other bytes.

7. 74 Next 8 bytes contain a binary floating point number of the same format as for item type '73' excepting that the 1st 7 bytes are the mantissa and the exponent is in the 8th byte. This is BASIC's double precision floating point format.

4. A file may be subdivided into records, either all of the same length (MF file) or of varying lengths (MU file).

5. Maximum length of a file record is 4095 bytes. This includes all record control, item control and data bytes.

6. If the file is divided into records not all of the same length (a MU file), then each record of the file starts with the SOR item automatically supplied by BASIC.

7. Successive records in the file may contain differing numbers of items. This will occur where the programmer has multiple record types within the file. For files with fixed length records, care must be taken to avoid record overflow.

8. Relatively positioned items within records of the file may differ as to type from one record to another. This will occur where the programmer has multiple record types within the file.

9. An I/O link to and/or from a marked item file is created by the BASIC statement OPEN with the ft parameter = MI, MU or MF.

10. Via the GET statement, the contents of marked item file items are moved into the BASIC variables specified in the IGEL.

11. Via the PUT statement, marked item file items are created from BASIC variables and/or BASIC expressions specified in the IGEL.

12. BASIC statement CLOSE terminates an I/O link between the program and a marked-item file.

13. No disk space is skipped between successive items or records of a marked item file. However, SOR and fill items are inserted as necessary.

8.7. OPEN

DISK BASIC's OPEN statement has been modified to handle the

following formats:

1. OPEN m,fan,filespec
2. OPEN m,fan,filespec,len
3. OPEN m,fan,filespec,ft
4. OPEN m,fan,filespec,ft,len

where:

8.7.1. See glossary for fan and filespec definitions. Examples of the four formats:

```
OPEN "I",1,"XXX/DAT:1"
OPEN "R",2,"XXX/DAT",128
OPEN "O",1,"XXX/DAT:0","MU"
OPEN "D",3,"XXX/DAT","MF",71
```

8.7.2. Format 1 above is used for print/input and field item files. Format 2 is used for field item files. Format 3 is used for FI, MI and MU files. Format 4 is used for MU, MF and FF files.

8.7.3. **m** specifies the operational mode for the filearea and is an expression evaluating to a string equal to one of the following:

1. **I** The filearea is open to the file for input operations only (INPUT if ft not specified - GET if ft specified). The filearea is positioned to the start of the file.
2. **O** If the file does not exist, it is created. The filearea is opened to the file for output operations only (PRINT if ft not specified - PUT if ft specified). EOF is set = 0, and the filearea is positioned at EOF.
3. **E** Same as "O" except EOF is not changed. This allows addition to an existing sequential file.
4. **R** If the file does not exist, it is created. The filearea is opened to the file for GET and/or PUT operations. EOF is not changed, file is positioned as for I. If a subsequent PUT specifies a record at or beyond EOF, the file is automatically extended to include that record.

5. **D** Same as **R** except that the file must already exist and a **PUT** for a record at or beyond **EOF** is treated as an error condition.

8.7.4. ft Specifies the file type and is an expression evaluating to a string equal to one of the following:

1. **FI** A fixed item file not record segmented. **len** must not be specified.
2. **FF** A fixed item file of fixed length records. **len** must be specified.
3. **MI** A marked item file not segmented into records. **len** must not be specified. Items within a **MI** file cannot be updated.
4. **MU** A marked item file segmented into records of varying lengths, where the length is determined by searching for either **EOF** or the next record's **SOR** item. **len** is optional and if specified is used as a maximum allowable length for the **MU** file's records. **AMU** file record may be updated provided the record length is not increased beyond its original value. If the record is shortened, it is filled out with fill items.
5. **MF** A marked item file segmented into fixed length records. **len** must be specified.

8.7.5. If **ft** is specified, the following apply:

1. If a **GET** statement is to actually transfer data from the file to **BASIC** variables, then the **GET** statement must specify either **IGEL** or **IGELSN**.
2. If a **PUT** statement is to actually transfer data from **BASIC** variables or expressions, then the **put** statement must specify either **IGEL** or **IGELSN**.
3. **BASIC** statement **FIELD** must not be used.
4. The program must not alter information within the filearea's **I/O** buffer, and must not rely upon values in that buffer or in the **LRECL**, **NEXT** or **EOF** fields of the **FCB**.

8.7.6. If **ft** is not specified and **m** = **R** or **D**, the following apply:

1. The file is a field item (random) file with specifications the same as for Model I **TRSDOS** 2.3 (Model III **TRSDOS** 1.3) except as otherwise noted.
2. **FIELD** statements must be used for proper overlay of **BASIC** variables into the filearea's buffer. **FIELD** can process 256 byte records though any one string defined therein is limited in length to 255 characters. The number of bytes defined by a **FIELD** statement is normally equal to **len**, should not exceed **len** and must not exceed 256.
3. **GET/PUT** statements must not specify either **IGEL** or **IGELSN**.
4. If **len** is not specified, **len** is assumed equal to 256.

5. len must be a value from 1 to 256. If len is less than 256, then BASIC must have been initialized explicitly specifying the filearea count suffixed with the character V (see section 7.3).

8.7.7. len An expression evaluating to an integer between 1 and 256 for field item files and between 1 and 4095 for fixed item and marked item files. For field item, FF or NY files, len is the standard length for records of the file. For MU files, len is the maximum length allowed for records of the file. Currently, the file's FPDE does not carry the correct len (LRECL) value; so the len value, explicit or implied, supplied at OPEN is always used. Checks on len are done during GET and PUT. For MF and MU files, the programmer must allow for the following extra bytes in the len calculations:

1. 1 byte for each item (primary item control byte)
2. 1 byte for each string actually containing more than 127 chars.

For MU files, the programmer must allow for the SOR item byte at each record's start.

The number of bytes assigned to a marked file item equals the number of marker (or control) bytes (1 or 2) plus the number of bytes used by BASIC to contain the string or the numeric:

1. Strings: one or two marker bytes plus the actual string length, allowing for truncation due to expression prefix. The second marker byte is used only if the string length is greater than 127 bytes.
2. Integers: 1 marker byte plus 2 bytes.
3. Single precision floating point: 1 marker byte plus 4 bytes.
4. Double precision floating point: 1 marker byte plus 8 bytes.

For fixed item files, the number of bytes assigned to each item is determined from the IGEL as:

1. For strings, for (len)\$ and for (len)#, the number specified by the expression prefix.
2. Integers: 2 bytes.
3. Single precision floating point: 4 bytes.
4. Double precision floating point: 8 bytes.

8.7.8. If the EOF in the FCB is modified by OPEN, a subsequent CLOSE or PUT,fan,&& statement will update the new EOF into the FPDE even though no PRINT or PUT statement was executed.

8.8. GET

DISK BASIC's GET statement has been modified to handle the following formats:

1. GET fan Up is null)
2. GET fan,fp 3. GET fan,fp,IGELSN 4. GET fan,fp „ IGEL

where:

8.8.1. fan and IGELSN are defined in the glossary. fp is defined in section 8.4.1 and IGEL in section 8.4.2. Examples of the 4 formats above are:

```
GET 1
GET 1,30
GET 1,!X,1000
GET 1,,,X%,Y!,Z#,(20)A$;
```

8.8.2. On successful completion of the GET statement, the filearea is left positioned at:

1. For marked item file ops, the next item of file.
2. For fixed item file ops, the next byte of the file.
3. For field item file ops, the next record of the file.

8.8.3. If FOR or EOF encountered:

1. For field item file ops, the filearea buffer is set to binary zeroes; thus giving binary zero value to all data subsequently referenced. No error occurs.
2. For marked item and fixed item file ops, an error occurs.

8.8.4. If an error is encountered during GET processing, the filearea control data is reset to the state existing prior to the GET statement. The resulting contents of the variables-named in the IGEL or FIELD are indeterminate. After error correction, the statement may be executed again.

8.8.5. If the GET statement specifies IGEL or IGELSN, then successive file items are processed into successively named variables of the IGEL. For marked file ops:

1. If an IGEL expression is null, the corresponding file item is bypassed.
2. An IGEL expression prefix can be used to limit the number of characters for the string variable. If the file item has less characters, the string length is set to the lesser value. If the file item has more characters, the excess characters on the right are bypassed and are not passed to the variable.
3. As fill items are encountered, they are bypassed.
4. Type-mismatch (TM) error occurs if the named variable and the file item are type incompatible.

5. For a record segmented file, a GET for the first item(s) may be followed by a PUT for the rest of the item(s).
6. For a record segmented file, record overflow error occurs if GET finds insufficient items in the record.
7. Except for the limiting effect of the expression prefix, strings are passed from the file to the variable as is. There is no leading blank suppression.

For fixed item file ops:

1. For each named string variable, the number of characters specified in the expression prefix is transferred from the file to the string area.
2. For record segmented files, 'RECORD OVERFLOW' error occurs if GET finds insufficient bytes in the record.
3. GETS and PUTS for successive data may follow one another at will providing:
 1. The user keeps good track of the current position within the record.
 2. Record boundaries are observed for a record segmented file.

For marked item and fixed item files:

The input of a record's items may be spread across two or more GETS.

8.8.6. The GET statement of the forms:

```
GET fan,!$rba
GET fan,!$%
```

allows the programmer to position the file for the next GET, INPUT, PUT or PRINT statement for that file area. No data transfer is done by this GET statement. 1\$% means the current value of EOF is to be used as the RBA value. Statements of this form mark REMRA and REMBA invalid. Examples:

```
GET 1,!$2550      positions the file to RBA 2550
GET 1,!$X         positions the file to the RBA value in X
GET 2,!$%         positions the file to EOF
```

8.9. PUT

DISK BASIC statement PUT is modified to handle the following formats:

1. PUT fan (fp = null)
2. PUT fan,fp
3. PUT fan,fp,IGELSN
4. PUT fan,fp,,IGEL

where:

8.9.1. fan and IGELSN are defined in the glossary. fp is defined in section 8.4.1 and IGEL in section 8.4.2. Example codings of these 4 formats are:

```
PUT 2
PUT 1,X
PUT 3,,1060
PUT 1,RN!,,(20)A$,B%,C!,D#;
```

8.9.2. On successful completion of the PUT statement, the filearea is left positioned as done for GET.

8.9.3. If an error is encountered during PUT processing, the filearea control data is reset to the state existing prior to the PUT statement. The resulting data in the file is indeterminate, and will probably cause errors to occur upon a subsequent GET. This should be a problem only when updating existing records, and if possible a subsequent PUT for that record should be issued after the error condition has been corrected. To reduce the occasions of file damage, when the file is opened m = R or D, the IGEL is processed once in it's entirety to catch non-I/O errors and then again to do the actual file update.

8.9.4. If PUT specifies IGEL or IGELSN, then the value of successive IGEL expressions are sent to successive items of the file. For marked item file ops:

1. SOR and fill items are inserted into the file automatically if and when necessary.
2. An IGEL expression may be anything legal on the right side of the equation in a let statement, excepting functions referencing a filearea.
3. Except for the limiting effect of the IGEL expression prefix, the resulting string is sent to the file as is.
4. Numeric literals or expressions are sent to the file as the BASIC numeric type they convert to internally in BASIC.
5. For fixed length records and updated variable length records, each PUT statement replaces that portion of the record from the PUT's file positioning through the end of the record, using fill items if and as necessary. ******* CAUTION** Any items previously existing in relative position in the record higher than the last item written by the PUT action are lost, as all of the record's disk space from the last item of the PUT to the end of record now contain fill items.

6. The maximum theoretical sum of bytes for a record (the sum of bytes used for control, for numeric data and for strings) can exceed len (defined in OPEN, section 8.7) so long as the actual number of bytes used during the record's PUT(s) does not exceed len.

For fixed item file ops:

For each string variable, the number of characters specified in the required expression prefix is transferred from the variable to the file by padding with blanks or truncating on the right done as necessary.

8.9.5. For marked item and fixed item files:

1. 1. The output of a record's items may be spread over two or more PUT statements.

2. 2. Data is moved into the filearea's buffer, but is not actually written to disk until one of the following occurs:

1. The filearea is closed.

2. The buffer is needed to contain data from another part of the file.

3. A 'PUT fan,&' or a 'PUT fan,&&' statement is executed.

3. 'RECORD OVERFLOW' error occurs if the allowable record length is exceeded.

4. See OPEN (section 8.7.7) for discussion of the number of bytes used by numeric file items.

8.9.6. The PUT statement of the form:

PUT fan,&

allows the programmer to force the write of the filearea's buffer to disk if that buffer contains data not yet written to disk. If the buffer has no such data, the statement is ignored. The programmer must remember that actual data writes to disk for marked item, fixed item and field item (where len less than 256) files are not necessarily done at PUT time, under the assumption that more write data may yet appear in the buffer. 'PUT fan,&' forces this pending data out to disk, and should be used whenever any of the following conditions exist:

1. It will be some time before the file area will be used again, but the programmer does not want to issue CLOSE.

2. Proper interaction with other fileareas depends upon the pending data being on the disk.

3. The data is very important.

The file area's file positioning is not affected by the PUT fan,& function.
Example:

```
PUT 3,&
```

8.9.7. The PUT statement of the form:

```
PUT fan,&&
```

allows the programmer to force the write into the directory of the EOF currently in the filearea's control data. This special PUT will save the programmer the necessity of doing a LOC(fan)1 function to remember the current file positioning, a CLOSE to cause EOF write into the directory, an OPEN to reestablish the link to the file, and a positioning GET or PUT to position the filearea back to where it was. Before actually writing the EOF to the directory, the PUT fan,&& function performs a PUT fan,& function. The filearea's file positioning is not altered by the PUT fan,&& function.

Example:

```
PUT 2,&&
```

8.9.8. The PUT statement of the forms:

```
PUT fan,!$RBA  
PUT fan,!$%
```

function identical to that for GET (see section 8.8.6). 8.9.9. The PUT statement of the form:

```
PUT fan,!#rba
```

causes the file's EOF to be set to the value of the expression rba, which must evaluate to a RBA. Nothing else is changed for that filearea. Remember, a CLOSE or a PUT fan,&& statement must be executed to force the write of the new EOF into the file's FPDE. Example:

```
PUT 2,1#2000
```

causes the EOF in filearea 2's control data to be set to 2000.

8.10. REMRA and REMBA

Within each filearea's control data, BASIC saves two additional relative file location values:

1. 1. REMRA Membered Record Address.
2. 2. REMBA Membered Byte Address.

where:

1. The ONLY places where REMRA is used is (1) to position the file when the GET or PUT statement has fp = # (see section 8.4.1.3) and (2) in the LOC(fan)\$, LOC(fan)! and LOC(1)# functions (see section 8.12).

2. The ONLY place where REMBA is used is to position the file when the GET or PUT statement has fp = \$ (see section 8.4.1.4).
3. Both REMRA and REMBA are in RBA format.
4. Each OPEN statement and each GET or PUT statement with rp = !RBA or !\$% marks both REMRA and REMBA as invalid.
5. Each INPUT and PRINT statement sets REMRA to the file position existing at the start of the statement execution. REMBA is not used for print/ input file ops.
6. Each GET or PUT statement with fp = null, rn, !rba, !% or * (for *, only if REMRA is invalid at statement start or if the file is not record segmented) sets REMRA = to the file positioning resulting from that fp value.
7. Each GET or PUT statement with fp = null, rn, !rba, !% or * sets REMBA = to the file positioning resulting from that fp value.
8. Don't let the concepts of REMRA and REMBA puzzle you too much. As stated above, there are only two places where REMRA is used (when fp = # and for the LOC functions) and only one where REMBA is used (when fp = \$). If you never use partial record I/O, then REMRA and REMBA are always the same. The most common use will be in executing a PUT (with fp = #) for the record just read.

8.11. Pseudo FIELD Function

For fixed item and marked item files, the FIELD statement is not legal. However, there are times when the programmer may want to set the strings associated with an IGEL to their specified lengths and keep them that way by using LSETs and RSETs. The user could do this by using the STRING\$ function. Another way is to use the pseudo FIELD function having the following formats:

1. GET fan,%,IGELSN
2. GET fan,%,,IGEL
3. PUT fan,%,IGELSN
4. PUT fan,%,,IGEL

where:

1. fan and IGELSN are defined in the glossary and IGEL is defined in section 8.4.2.
2. fan specification is required for text format protocol only. Whether the filearea is open or what it is opened for is not of concern to this pseudo FIELD function; this function is only concerned with the IGEL and does not alter the filearea in any way.
3. The IGEL is processed:
 1. Numeric variables are left unchanged.
 2. Expressions of the form (len)\$ and (lenh are bypassed.

3. String variables in the IGEL must be prefixed.

4. String variables are assigned length = to the IGEL expression prefix and either truncated or padded on the right with blanks as necessary. Aside from the padding or truncation, the string contents are not changed. However, if the string is not currently in the string area, it is moved there. Subsequently, LSET and RSET may be used to move data into these strings.

4. Example:

```
PUT 2,%,IX%,(30)A$,DP#,(10)B$;
```

causes string A\$ and B\$ to be made into strings 30 and 10 characters in length respectively, being padded with spaces or truncated on the right as necessary. No data is transferred to the file and file positioning is not changed.

8.12. LOC Function

NEWDOS/80 DISK BASIC has a LOC function defined as follows:

1. LOC(fan) where fan is a file area number, 1 - 15, of a filearea opened for field item, MF or FF file operations. This function returns an integer 1 - 32767 = the number of the previous record GET/PUT for that file area. 0 = none or REMRA invalid. Example:

```
PUT 1,34
X = LOC(1)
```

results in X have the value 34.

2. LOC (fan)\$ For record segmented files, this function returns -1 (IF statement true) if the start of the next record (if REMRA valid) or the current file position (if REMRA invalid) is greater than or equal to EOF, and returns 0 (IF statement false) if less than EOF. For non-record segmented files and print/input files, this function returns -1 (IF statement true) if the current file positioning is greater than or equal to EOF, and returns 0 (IF statement false) if less than EOF. LOC (fan)\$ differs from function EOF in that EOF tests only for exactly at EOF.

Example:

```
IF LOC(1)$ THEN END
```

ends the program execution if the next record is located at or beyond the file's EOF.

3. LOC (fan)% Returns an RBA equal to the file's EOF. Example, suppose the file contains 3142 bytes:

```
X = LOC(1)%
```

will result in X having the value 3142.

4. LOC (fan)! For record segmented files, this function returns a RBA value equal to:

1. If REMRA valid, the location of the file's next record.
2. If REMRA invalid, the current file position.

For non-record segmented files and print/input files, this function returns an RBA equal to the current file position.

Example, if the latest fully or partially processed record for filearea 1 starts at relative file position 1667 and the next record starts at relative file position 17\$1, then

```
X = LOC(1)!
```

will set X equal to 1701.

5. LOC(fan)# Returns an RBA value equal to REMRA. Error if REMRA currently invalid. Example, see above example:

```
X = LOC(1)#
```

will set X = 1667.

Use of LOC(fan)! and/or LOC(fan)# allows the programmer to obtain the file position of a group of items (non-record segmented file) or a record (record segmented file), remember it for future use, and then at a future time, reposition the file to that data via either fp = !rba or fp = !\$rba. This allows programmers to build index files that index into all types of files for random accessing.

8.13. I/O Error Recovery

The operation of the DISK BASIC statements PRINT, PUT, INPUT, and GET has been altered such that if an error occurs during statement processing, the filearea control data is left unchanged by that statement. This allows the user/programmer more options when an error occurs. Examples:

1. The program is outputting to a sequential print/input file. 'DISK FULL' error occurs. EOF is returned to where it was at the statement beginning; the file can then be closed, and if no other files are open on that drive, another diskette can be mounted, a new file opened for the same file area, and then the statement in error executed again to continue processing. Later input processing can then process both files, using EOF on the first to trigger the shift to the 2nd.

2. The program is outputting to a MU file using two or more PUTS to output a single record. 'DISK FULL' error occurs on the 2nd PUT of the current record. EOF is reset to where it was at the error statement's beginning, not to record's beginning. Before switching to a new file, EOF must be set back to the record's beginning via the following two statements:

```
X!=LOC(fan)#: PUT fan,!#X!
```

Then the file area may be closed, a new diskette mounted, the filearea reopened, and processing continued back at the beginning for the record (not to the beginning of the PUT). Since a MU file must always start with an SOR item, if two MU files are used in concatenation, the 1st cannot end with a partial record in anticipation of the next containing the rest of the record.

***** The user/programmer must use extreme caution in swapping diskettes on one drive or in swapping a given diskette to another drive when more than the error filearea is open for the original drive.

Also to be remembered is that though the filearea control data is restored to what it was at the statement beginning, the file data associated with a PUT is indeterminate, and the contents of the variables receiving data on a GET is also indeterminate.

In order to facilitate error recovery and coding in general, BASIC uses a separate control area to perform the GET, PUT or other filearea related operations, leaving the filearea's control data unchanged until the operation completes without error. In NEWDOS80 there is only one temporary control area; a function using a filearea CANNOT be nested within another function using a filearea, even if both file areas are the same. For example, the two statements given above CANNOT be combined into one as:

```
PUT f8n,1#LOC(fan)#
```

8.14. Additional notes about NEWDOS/80 DISK BASIC I/O

1. For marked item and fixed item files, the programmer GETS or PUTS an item-group of data at one time. The only limitations on the amount of data transmitted are file size and, if applicable, record size. Logical records can be any length between 1 and 4095 bytes. The programmer should never refer to the filearea buffer(s), as the contents at any time are unpredictable. ***** **WARNING** ***** If the program alters data in the filearea's buffer when a file is opened for anything other than field item operations where FIELD was and is legal, the results are unpredictable and usually disastrous. Extreme caution must be used to avoid the file damaging situations where FIELD statements have been legally used, then that filearea used for I/O where FIELD is not legal but RSET or LSET functions continue to be used for one or more FIELD defined strings for that filearea.

2. The special functions designed for field item file ops, (MKD\$, MKI\$, MKS\$, CVD, CVI, CVS, LSET, RSET, etc.) work as before. However, the use of MKD\$, MKI\$, MKS\$, CVD, CVI, and CVS may be dropped for marked item or fixed item file ops as GET and PUT will transmit numeric as well as string data.

3. For GET or PUT statements using either IGEL or IGELSN, the programmer must remember that any errors detected during IGEL processing will be recorded as an error occurring on the line containing the GET/PUT rather than on the actual text line of the IGEL.

4. To facilitate error detection for GET or PUT statements using IGELSN, the GET or PUT should be the only statement on its text line.

5. A file can be updated only if it can be opened R or D. MI and print/input files cannot be updated, though of course they may be added onto. MU file records can be updated provided the new record length does not exceed the original length of the record. The last record of a MU file may be extended without this restriction.

6. Fileareas open for print/input files may have GET or PUT statements executed for them if the fp type is !\$rba, !\$%, !#rba, &, && or % .

7. BASIC functions (i.e., EOF, LOC, LOF, etc.) that use fan cannot exist within an IGEL or within OPEN, GET, PUT, CLOSE, PRINT (to disk) or INPUT (from disk) statements. This is a NEWDOS/80 restriction not existing in TRSDOS and is imposed by the error recovery operations (see section 8.13).

8. For disk files whose records can span two or more disk sectors (files whose record lengths are either not standard or do not divide into 256 evenly), the number of actual disk I/O's is increased up to 200% (as compared with files whose record lengths are standard and do divide into 256 evenly) when a record or item group actually has parts in two or more file sectors. The percent overall increase in disk I/O is approximately $(LEN/256)*200$ where LEN is the average length of records or item groups processed, and where $LEN < 256$. No approximation is given for $LEN > 256$.

9. ERROR CODES AND MESSAGES

9.1. DOS Error Codes and Messages

The following is a list of DOS error messages for NEWDOS/80 Version 2 corresponding to error codes placed in register A on a CALL or JP to 4409H. The codes are listed in both decimal and hexadecimal.

00	00	NO ERROR
01	01	BAD FILE DATA
02	02	SEEK ERROR DURING READ
03	03	LOST DATA DURING READ
04	04	PARITY ERROR DURING READ
05	05	DATA RECORD NOT FOUND DURING READ
06	06	TRIED TO READ LOCKED/DELETED RECORD
07	07	TRIED TO READ SYSTEM RECORD
08	08	DEVICE NOT AVAILABLE
09	09	UNDEFINED ERROR CODE
10	0A	SEEK ERROR DURING WRITE
11	0B	LOST DATA DURING WRITE
12	0C	PARITY ERROR DURING WRITE
13	0D	DATA RECORD NOT FOUND DURING WRITE
14	0E	WRITE FAULT ON DISK DRIVE
15	0F	WRITE PROTECTED DISKETTE
16	10	DEVICE NOT AVAILABLE
17	11	DIRECTORY READ ERROR
18	12	DIRECTORY WRITE ERROR
19	13	ILLEGAL FILE NAME
20	14	TRACK # TOO HIGH
21	15	ILLEGAL FUNCTION UNDER DOS-CALL
22	16	UNDEFINED ERROR CODE
23	17	UNDEFINED ERROR CODE
24	18	FILE NOT IN DIRECTORY
25	19	FILE ACCESS DENIED
26	1A	DIRECTORY SPACE FULL
27	1B	DISKETTE SPACE FULL
28	1C	END OF FILE ENCOUNTERED
29	1D	PAST END OF FILE
30	1E	DIRECTORY FULL. CAN'T EXTEND FILE
31	1F	PROGRAM NOT FOUND
32	20	ILLEGAL OR MISSING DRIVE #
33	21	NO DEVICE SPACE AVAILABLE
34	22	LOAD FILE FORMAT ERROR
35	23	MEMORY FAULT
36	24	TRIED TO LOAD READ ONLY MEMORY
37	25	ILLEGAL ACCESS TRIED TO PROTECTED FILE
38	26	FILE NOT OPEN
39	27	ILLEGAL INITIALIZATION DATA ON SYSTEM DISKETTE
40	28	ILLEGAL DISKETTE TRACK COUNT
41	29	ILLEGAL LOGICAL FILE
42	2A	ILLEGAL DOS FUNCTION
43	2B	ILLEGAL FUNCTION UNDER CHAINING
44	2C	BAD DIRECTORY DATA
45	2D	BAD FCB DATA

46	2E	SYSTEM PROGRAM NOT FOUND
47	2F	BAD PARAMETERS)
48	30	BAD FILESPEC
49	31	WRONG DISKETTE RECORD TYPE
50	32	BOOT READ ERROR
51	33	DOS FATAL ERROR
52	34	ILLEGAL KEYWORD OR SEPARATOR OR TERMINATOR
53	35	FILE ALREADY EXISTS
54	36	COMMAND TOO LONG
55	37	DISKETTE ACCESS DENIED
56	38	ILLEGAL MINI DOS FUNCTION
57	39	OPERATOR/PROGRAM/PARAMETER REQUIRE FUNCTION TERMINATION
58	3A	DATA COMPARE MISMATCH
59	3B	INSUFFICIENT MEMORY
60	3C	INCOMPATIBLE DRIVES OR DISKETTES
61	3D	ASE=N ATTRIBUTE. CAN'T EXTEND FILE
62	3E	CAN'T EXTEND FILE VIA READ

If the error code is not defined, UNKNOWN ERROR CODE message will be displayed.

SYS4/SYS is the DOS error message display module.

9.2. DISK BASIC Error Codes and Messages

In addition to the standard ROM BASIC LEVEL II error codes, the following DISK BASIC error codes are used:

51	FIELD OVERFLOW	68	TOO MANY FILES
52	INTERNAL ERROR	69	DISK WRITE PROTECTED
53	BAD FILE #	70	FILE ACCESS DENIED
54	FILE NOT FOUND	71	SEQ # OVERFLOW
55	BAD FILE MODE	72	RECORD OVERFLOW
56	FILE ALREADY OPEN	73	ILLEGAL TO EXTEND FILE
58	DOS ERROR	75	PREVIOUSLY DISPLAYED ERROR
59	FILE ALREADY EXISTS	76	CAN'T PROCESS LINE
62	DISK FULL	77	BAD FILE TYPE
63	INPUT PAST END	78	IGEL SYNTAX ERROR
64	BAD RECORD #	79	IGEL ITEM SYNTAX ERROR
65	BAD FILE NAME	80	BAD/ILLEGAL/MISSING IGEL ITEM PREFIX
66	MODE MISMATCH	82	BAD RECORD LENGTH
67	DIRECT STATEMENT IN FILE	83	STMT USES 2 FILE NAMES
		84	BAD FILE POSITIONING PARAM

SYS13/SYS is the module that displays DISK BASIC and ROM BASIC error messages. It is normally not in memory until needed. If an error code is generated for which there is no message, UNPRINTABLE ERROR is displayed.

10. GLOSSARY

This chapter contains the definitions of some of the terms used throughout the NEWDOS/80 documentation.

alpaha or alpha character

Used when referring to the set of characters A - Z and a - z.

alphanumeric

Used when referring to the set of characters A - Z, a - z and 0 - 9.

bit

The smallest accessible unit of main or diskette memory. A bit has a value of either 0 (meaning off) or 1 (meaning on). A group of 4 consecutive bits is known as a hexadecimal (or hex) digit, and a group of 8 consecutive bits is known as a byte. Whenever the documentation refers to a bit within a byte, the convention is bit 7 is the bit on the left and bit 0 is the bit on the right with the order of bits within a byte going left to right, 7 to 0. The concept holds for bits within a hex digit, left to right, 3 to 0.

boot see reset/power-on.

BOOT/SYS

One of the two control files required on every diskette used with NEWDOS/80. See section 5.1.

buffer

An area of main memory used to hold the contents of a sector read from disk or to hold the new contents of a sector being written to disk. Each open FCB has a 256 byte buffer assigned for this purpose. Byte mode disk I/O, such as is used for print/input, marked item, fixed item, and (if record length less than 256) field item files actually operates to and from the buffer with disk sector reads and writes being done when necessary, and not on each GET or PUT or PRINT or INPUT statement execution.

byte

The smallest addressable unit of main or diskette memory. A byte is composed of 8 bits. When the value of a byte is given, it is usually expressed as two hexadecimal digits. In NEWDOS/80 documentation, the words byte and character are used interchangeably even though character can have a more restrictive meaning.

chaining

Used in NEWDOS/80 to refer to the process of bringing keyboard input characters from a disk file known as a chain file. See section 4.3.

character

Used interchangeably with byte, but also used to refer to a byte containing a printable value.

close

In disk I/O, to close a FCB or a filearea means to dissolve the link between a program and a disk file created by the open function.

DEC Directory Entry Code

A one byte code used to specify a particular FDE and used by DOS to quickly locate that FDE in the directory. When an FCB is open, its 8th byte contains the DEC for the file's FPDE. Each FXDE contains in its 2nd byte the DEC for the preceding FDE for the same file, and each FPDE or FXDE whose 31st byte = 255 0FEH) contains in its 32nd byte the DEC of the next FXDE for the file. The format of the 8 bit DEC is:

rrrsssss

where sssss+2 = the relative number within the directory of the sector containing the FDE, and rrr times 32 (20H) equals the relative byte address within the sector of the FDE.

DIR/SYS see sections 5.1 and 5.6.

One of the two control files required on every diskette used with NEWDOS/80. DIR/SYS contains the directory for a diskette.

directory see sections 5.1 and 5.6.

In DOS, the directory refers to the contents of the file DIR/SYS that must be present on every diskette used by NEWDOS/80. The directory contains the control information specifying all files and the free or allocated state of all space on the diskette. If the directory is damaged or destroyed, the rest of the information on the diskette is usually, but not always, no longer available to the user.

DOS Disk Operating System

Though many thousands of programmers are quite capable of writing their programs to communicate directly with the diskette, it is almost always preferable to allow another program, or collection of programs, to act as an intermediary between the user program and the disk files the program uses. This intermediary is commonly called a DOS and serves to both structure and vastly simplify a program's I/O with the files it uses. Usually, as in NEWDOS and TRSDOS, the DOS functions are much more extensive such that the DOS becomes the primary control program in the computer and has available various other functions, other than disk I/O control, that it performs in response to commands, known as DOS commands (specified in chapter 2), or DOS calls (specified in chapter 3). In NEWDOS/80, the DOS operates in the 4000 - 51FFH region of main memory with some of its functions using the 5200 - 6FFFH region and the spooler running out of highest memory.

DOS-CALL or dos-call

Refers to the DOS state entered when a user program calls the DOS routine at 4419H (see sections 3.11 and 4.4) to execute a DOS command or a user program. There can be multi-levels of DOS-CALL state.

DOS command or doscmd

Refers to one of the built-in DOS functions described in chapter 2. DOS commands can be executed by keying in from the keyboard or through calls from the current executing program (see DOS-CALL).

EOF End Of File
 Of or pertaining to the end of a file. Some files have one or more specific EOF bytes that mark the end of a file (assembler source files use 1AH, BASIC non-ASCII text uses 3 consecutive bytes of zeroes, etc.); however, most files do not and rely entirely upon the EOF within the FCB or FPDE to indicate where the file ends. If a file is empty, EOF equals and if a file has 1324 bytes, the EOF value expressed as an RBA is 1324. Within a NEWDOS FCB, EOF is a three byte RBA value of the file's last byte+1. The EOF value stored in a file's FPDE is not in RBA format. See sections 5.7 (fpde bytes 4, 21 and 22) and 5.9 (FCB 9, 13 and 14).

EOL End Of Line
 Of or pertaining to the end of a line. For input data or a command, this is usually the ENTER character (0DH). For BASIC text, a zero byte ends a line. If the line does not have an explicit EOL character, then EOL means the line's last character + 1.

EOM
 Of or pertaining to the end of a message. The EOM character code is 03. EOM is used to end a message when that message end is not also the end of the line. When encountered, the EOM character is not displayed or printed nor is the display or printer advanced one character.

EOR End Of Record
 Of or pertaining to the end of a record. FOR is also the relative byte address within the file of the record's last byte + 1.

EOS End Of Statement
 Of or pertaining to the end of a statement. For BASIC text, a colon ends a statement.

extent element
 A two byte control element within a FPDE or FXDE specifying a 1 to 32 granule contiguous area of diskette storage assigned to the file. See section 5.7, FPDE 23rd-30th bytes.

fan file area number
 A fan is a BASIC expression evaluating to an integer (range 1 - 15) specifying which filearea is to be used for the current BASIC function.

FCB File Control Block.
 See section 5.9. A data area containing information controlling an I/O link between a program and a diskette file. The link is created by the open function, dissolved by the close function, and used by all other disk I/O functions including GET, PUT, PRINT, INPUT, LOC, etc. The FCB contains the NEXT and EOF fields, the buffer address, security information, record length, etc.

FDE File Directory Entry. See section 5.6.3.
 In NEWDOS, each sector of the directory file DIR/SYS, except for the first two, is divided into eight 32 byte control areas called FDEs. A FDE is either free (available for assignment) or in use as a FPDE or FXDE.

FF file
 A BASIC fixed item file segmented into records all of the same length.

FI file

A BASIC fixed item file that is not record segmented.

file or disk file or diskette file

A collection of data on a disk or diskette. A file may contain diskette control information (as do BOOT/SYS and DIR/SYS), a machine language executable program (as do SYS0/SYS, BASIC/CMD and SUPERZAP/CMD), a BASIC program (as does CHAINTST/BAS) or user data (such as mailing lists, payroll, inventory). Control data for all files is contained within the file DIR/SYS (see section 5.6) with each file being assigned one FPDE and zero or more FXDEs. A file must exist entirely on one diskette. Diskette space is allocated to a file as needed in units called granules.

filearea

An area of BASIC's system storage containing control information, a FCB and a 256 byte buffer. A filearea is used during disk file operations to maintain an I/O link between a file and the BASIC program. This I/O link is established by OPEN, used by PRINT, INPUT, GET, PUT, FIELD, EOF, LOF, LOC, etc., and dissolved by CLOSE. When 2 or more fileareas are open to the same file, each acts in ignorance of the others. A BASIC program may have open at any one time as many as 15 fileareas. The number of fileareas actually available to the BASIC program is specified when BASIC is activated (see section 7.2) with the default being 3.

field item file

This is a name used in NEWDOS/80 for what, in TRSDOS disk BASIC, is called a random file since all three types of files, field item, fixed item and marked item can be used either randomly or sequentially or both. Field item and fixed item files are essentially the same type of file; the main difference is in the type of I/O link, field item or fixed item, used. For field item files, the definition of the file items is done solely via the FIELD statement. Field item files are always segmented into records all of the same length, with that length being from 1 to 256 bytes.

file item

A unit of file storage zero or more bytes in length containing a numeric value or a character string.

filespec

This term is used in NEWDOS/80 to refer to the combination of file name, name extension, password and drive number used to specify a file in a DOS command, BASIC statement or an unopen FCB. Of the four elements, only file name is required. See section 2.1 for full definition of filespec.

fixed item file See section 8.4.

Fixed item and field item files are essentially the same type of file. The difference lies in the type of link, field item or fixed item, used in the file I/O. For fixed item file processing, the definition of the file items is entirely dependent upon the IGEL used in the GET or PUT statement. There are two types of fixed item files, FI and FF.

format

Aside from many other definitions of the word format, it is also the word used for the process that prepares a raw diskette for use under NEWDOS/80. This process magnetically structures the diskettes into tracks which are at the same time further sub-divided into 256 bytes

sectors. Depending on the drive type, the diskette will contain 35, 40, 77 or 80 tracks, and depending upon the drive type and recording density, each track will contain 10, 17, 18 or 26 sectors.

- fp** file positioning
See section 8.4.1. fp refers to the second parameter of a GET or PUT statement. fp specifies the file positioning to be done during the file positioning phase that precedes the data transfer phase, if any, of a GET or PUT statement.
- FPDE** File Primary Directory Entry
See section 5.7 for FPDE specification. A FPDE is created in the diskette directory whenever a file is created. If a file exists on a diskette, there will always be a FPDE for it in the directory. The FPDE contains the file name, extension, passwords, protection level, EOF, the first 4 extent elements and other information. When a file is killed, the FPDE and any associated FXDEs are dissolved.
- FRDE** File Extended Directory Entry
See section 5.8 for FXDE specification. Whenever the number of extent elements needed to account for a file's diskette space exceeds four, one or more FXDEs are created in the directory to hold the extra extent elements, a maximum of four per FXDE. If a file has FXDEs, they are accessed via the FPDE. As a file's diskette space requirements change, FXDEs are created or dissolved as necessary, and when a file is killed, all FXDES associated with that file are dissolved.
- GAT** Granule Allocation Table
See section 5.6.1. The GAT is that portion of the directory's 1st sector (known as the GAT sector) wherein the free or allocated status of each granule is accounted for.
- granule**
The smallest unit of diskette storage allocatable to or de-allocatable from a file. When a file needs diskette space, one or more granules is allocated. For NEWDOS/80 a granule consists of 5 sectors equaling 1280 bytes.
- hash code**
Hash code as used in the DOS refers to a one byte encode of a file's name and extension used during open to rapidly find the file's FPDE in the directory. Hash codes are stored in the HIT sector, see section 5.6.2.
- hexadecimal or hex**
A numbering system using 16 digits, rather than 10 used by the decimal system. The digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F. The reason for the use of hexadecimal as opposed to decimal is that a hexadecimal digit is an easy way to express the value of 4 consecutive bits, where the following table defines the correspondence between a hexadecimal digit and four binary bits.

0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

Hexadecimal representation of disk, file or main memory locations and contents are widely used in the computer industry. Though some users can get by without learning anything of hexadecimal, we strongly recommend that users learn the rudiments, at least enough to understand the SUPERZAP and DEBUG displays. Throughout NEWDOS/80 and its documentation a hexadecimal numeric value is expressed with a suffixed H character (i.e., 13 = 0H or 256 = 100H) unless otherwise specified.

HIMEM

Refers (1) to the address of the highest usable main memory location, (2) to the 2 byte main memory area (Model I locations 4049H - 404AH and Model III locations 4411H - 4412H) where the HIMEM value is stored and (3) to the name of a DOS command (see section 2.25). Main memory above HIMEM is either non-existent or is reserved for other uses. All user Z-80 code programs should be coded to observe HIMEM.

HIT Hash code Index Table

See section 5.6.2. That portion of the directory's second sector (also known as the HIT sector) that contains the hash codes for all files on the diskette. Instead of searching the entire directory for a file's FPDE during open, DOS computes the hash code from the file name and extension, looks it up in the HIT sector and then goes directly to the sector containing the FPDE.

I/O input and/or output

I/O link or I/O path

Actual disk I/O between a disk file and main memory is done via an I/O link (also known as an I/O path) created by open, dissolved by close, and used by GET, PUT, PRINT, INPUT, LOC, EOF, etc. While the link is open, the controlling information for the link is contained in a FCB or filearea (which contains a FCB). Multiple links to the same file can be open at the same time with each link knowing nothing of the others. An I/O link remembers the position in the file where it is operating; thus multiple links can be operating on the same file at the same time. However, be careful as, remember, each I/O link knows nothing of the other's actions.

IGEL Item Group Expression List

See section 8.4.2. An IGEL is a list of BASIC expressions corresponding to a group of file items during the execution of a GET or PUT statement used in fixed item or marked item file processing.

IGEL expression See section 8.2.3.

An IGEL expression (usually but not always a BASIC variable) is that part of an IGEL corresponding to a file item. For each file item processed in a fixed item or marked item file GET or PUT statement, there is a corresponding IGEL expression in the IGEL.

IGELSN IGEL Sequence Number

The line number (also known as sequence number) of the BASIC text line containing the first or only line of the IGEL to be processed by the current GET or PUT statement. If used, the IGELSN is the 3rd parameter of the GET or PUT statement. An IGELSN is used in a fixed item or marked item GET or PUT statement whenever the GET or PUT statement itself does not contain the IGEL, and this usually occurs when the same IGEL is used by two or more GET and/or PUT statements.

item group
 A group of zero or more file items. In BASIC, an item group is the zero or more file items processed by an individual INPUT, PRINT, GET or PUT statement and is most commonly equivalent to a logical record.

len See section 8.7.7 and see LRECL
 The parameter in a BASIC OPEN statement that specifies either the standard or the maximum record length.

logical record
 A group of meaningful related file items. Though file data is physically ordered on the diskettes into sectors, the programmer usually deals with data groupings that are logically related and grouped, rather than physically related and grouped. Thus, when data is read from or written to a file, it is usually done so in logical record units.

LRECL Logical RECORD Length
 This is the standard or maximum length in bytes for records of a file. For non-BASIC files LRECL is 0 - 255 (with 0 meaning 256) and, is stored in the FPDE's 4th byte (though never used) and the FCB's 10th byte. In BASIC, LRECL is equivalent to len (see section 8.7.7).

lump
 refers to a division of diskette space as that space is accounted for in the diskette directory. Each of the first 192 bytes in the GAT sector contains either space allocation or lockout information for one lump where, depending on the number of granules per lump, each bit within the byte is either unused or specifies the allocated/free or non-existent/ existent state of one of the lump's granules. This definition was coined for use with NEWDOS/80 Version 2 to avoid using the words track and cylinder. See sections 5.6.1 and 5.7 (23-30th byte discussion).

marked item file see section 8.6.
 A file in which each file item is identified as to length and type by a prefixed marker byte. A marked item file is distinctly different from a print/input, field item or fixed item file. The three types of marked item file are MI, MU and MF.

MF file
 A marked item file that is segmented into records all of the same length. MI file A marked item file that is not record segmented.

ms millisecond

MU file
 A marked item file that is segmented into records of differing lengths.

null
 The absence of a parameter or expression. When parameters are separated by commas, back to back commas („) indicate a null.

null character

A character or byte with value = 0.

null string

A string or an expression evaluating to a string zero characters in length.

open

In disk I/O, to open a FCB or a filearea is to establish a link between the program and a disk file, using the FCB or filearea (which contains a FCB) to hold the link's control data. Though it is quite common to say that a file is opened, it is more correct to say that a FCB or filearea is opened for there is nothing in the disk file indicating open or closed state or the number of links opened to it as more than one FCB or filearea may be open to a given file at the same time. The link established by open remains until dissolved by the close function. It is the link that determines the type of I/O done with a file and where in the file. Thus, if differently specified links are established to the same file to exist concurrently, the same file data can be used but interpreted differently by each of the different links.

partial record I/O

Refers to instances where I/O is done in partial rather than full logical records. In BASIC, GETS and PUTS for marked-item and fixed-item files may operate in this manner though they usually operate in whole record I/O mode.

patch see zap.

power-on/reset See reset/power-on

print/input file

A disk file written to by PRINT statements and read by INPUT statements.

record segmented file

A type of file that can be broken down into logical records by BASIC. These file types are field item, FF , MF and MU.

REMBB REMembered Byte Address See section 8.10.

REMRA REMembered Record Address See section 8.10.

RBA Relative Byte Address

A method of addressing within a file, record, control block, etc. where addressing starts at 0 rather than 1. The first byte of the unit has RBA = 0. The nth byte in the unit has RBA value = n-1. In NEWDOS, RBA is used to express EOF and NEXT in the FCB; this use of RBAs in the FCB is major difference between NEWDOS and the old versions of TRSDOS. In BASIC, RBA is used in file positioning (see section 8.4.1) where, in fp = !rba, !\$rba or !#rba, rba is defined to be a BASIC expression evaluating to a number between 0 and 16,777,215 and represents a relative byte position from the beginning of the file.

reset/power-on also known as boot.

refers to the automatic computer execution that occurs whenever the computer's reset button is pressed or when the computer is powered up. In reality, you must never have diskettes in any drives when you power up the computer. After the power up, put the system diskette in drive 0 and press reset. For the most part, NEWDOS/80 treats a reset after power-on the same as a reset at any other time. There are some

differences, however, with the most notably being the date and time settings that occur.

During a reset/power-on, the ROM's bootstrap routine receives computer control from the hardware reset logic and reads the first sector of the diskette mounted in drive 0 into the DOS system buffer (4200H - 42FFH on the model I and 4300H - 43FFH on the model III). That 256 bytes contains NEWDOS's bootstrap routine which receives computer control from the ROM and then reads into main memory a fresh copy of NEWDOS/80's main memory resident module SYS0/SYS. Execution control is then passed to SYS0's initialization routines in the DOS overlay area. Using the current SYSTEM and PDRIVE specifications, NEWDOS/80 is initialized. When this is completed, either NEWDOS/80 READY is displayed or DOS commences the execution of the AUTO (see section 2.4) specified DOS command.

sector

For NEWDOS/80, diskette data storage is physically done in groups of 256 bytes called sectors. Actual diskette reads and writes are done by whole sectors, usually a single sector at one time.

SOR Start Of Record

Of or pertaining to the start of a record. All records of a MU file start with a SOR item, a 70H byte.

track

The unit of diskette storage a disk drive read/write head passes over during one revolution of the diskette. A diskette is divided magnetically into a number of concentric tracks during format (35 is standard on the model I, 40 on the model III). Format also divides each track magnetically into 256 byte sectors which will subsequently contain data of any and all kinds.

user segmented file

A type of file which cannot be broken down into logical records by BASIC. These file types are FI and MI. If these file types are to be segmented into records, it is done so solely by the programmer without BASIC's knowledge.

vice

Means 'instead of or 'in place of'.

whole record I/O

Whole record I/O is when an entire logical record is read or written during the execution of a single INPUT, PRINT, GET and PUT statement. This is the normal procedure for those statements. See partial record I/O.

zap

To alter data or program executable code without recompilation. See section 11.

11. ERROR REPORTING, INCOMPATIBILITY HANDLING, AND PATCHING

11.1. Introduction

As with previous NEWDOS versions, NEWDOS/80 Version 2 will contain errors not presently known, will receive minor enhancements as the months pass, and has incompatibilities with other DOSs including earlier versions of NEWDOS. Where possible and economically feasible, patches (zaps) will be issued to correct the errors, provide the enhancements and, in selected cases, relieve the incompatibilities.

Apparat relies heavily on the NEWDOS/80 users to find and inform Apparat of NEWDOS errors and incompatibilities. Over half of the zaps generated for NEWDOS/80 Version 1 were a direct result of an error properly reported. In some cases, the user had to report the error more than once before Apparat either paid attention or finally found the error. Reported errors may or may not be fixed, depending upon the seriousness, the magnitude and the amount of zap area available in the affected modules. If an error is not to be fixed, Apparat will, in a comment zap, report the error and announce that it will not be fixed.

11.2. Incompatibility Handling

NEWDOS/80 is a different DOS from TRSDOS, VTOS, LDOS, DOSPLUS and others; therefore many user programs will not operate on NEWDOS/80 without some modification. For any particular program, the best thing is to try that program out with NEWDOS/80; be sure you do not use valued file data in these tests. In the past, Apparat has tried to create and distribute the necessary patches to commonly used, commercially sold programs, but this proved unworkable for a number of reasons.

1. Apparat was not notified by program manufacturers of a pending release of a new program and of its actual incompatibility with NEWDOS/80. The discovery of the incompatibility always came from the users. This is not a criticism, only a statement of fact.
2. Apparat did not and does not have the personnel resources to research each incompatibility problem and to generate the necessary zaps to the non-NEWDOS/80 programs.
3. The mailing of zaps to all registered NEWDOS/80 owners was delayed until a number of zaps were available, a delay usually of months, though Apparat would mail out the latest zaps to individuals on request. It would be much better if the necessary incompatibility zaps were sent out along with the non-NEWDOS/80 program. Apparat, in the past, did not make an effort to send the zaps to the manufacturers to include with their programs, and for this we apologize.

For NEWDOS/80 Version 2, Apparat will still issue compatibility zaps for some application programs, **but fundamentally Apparat will rely on the creator and/or distributors of non-NEWDOS/80 programs to produce and distribute the zaps necessary, if any, to run those programs with NEWDOS/80.** To assist in this effort, Apparat offers a free copy of NEWDOS/80 to business firms that

produce software products to be used on NEWDOS/80, provided these products are advertised in a major publication (NEWDOS/80 need not be mentioned in the advertisement).

11.3. Reporting of NEWDOS/80 Errors and Incompatibilities

To reduce confusion, frustration, cost and wasted time, Apparat requires that the following be done:

1. Read and understand the applicable documentation.
2. For errors, assure that language programs using NEWDOS/80 are interfacing correctly. Apparat does not check out programs other than what it creates.
3. Assure that all outstanding mandatory zaps have been applied to your NEWDOS/80 system or user programs.
4. Run the circumstances resulting in the NEWDOS/80 error or incompatibility many times under varying conditions (if possible).
5. Precisely and concisely write up the error circumstances and send, along with applicable diskettes, to:

Apparat, Inc.
4401 S. Tamarac Parkway
Denver, CO 80237

6. Include your NEWDOS/80 registration number.
7. Include copies of the diskettes (as gifts to Apparat) containing the all the modules involved in the error or incompatibility. Apparat will destroy the diskettes' contents, including any copies made of them, when done with the error study.
8. **DO NOT PHONE Apparat directly.** Phone answering personnel are not technically knowledgeable of NEWDOS.
9. **DO NOT INCLUDE** product orders or other requests with your error report.

11.4. Format of NEWDOS/80 Zaps

In NEWDOS/80, zaps (patches) are manually applied by using the program SUPERZAP discussed in section 6.1. The user should study section 6.1 to learn how to use SUPERZAP, but if he/she prefers not to do that, enough information will be provided in this chapter to scrape by.

Though SUPERZAP is a somewhat cumbersome method of applying zaps, this method does have the advantage of forcing the users to learn how to use SUPERZAP and

gives them confidence in using that program they would otherwise not have acquired. Sooner or later, everybody needs to use SUPERZAP to help repair damaged disk files, and when this emergency arises, the more experience the user has had with SUPERZAP, the better.

NEWDOS/80 zaps are consecutively numbered and are dated with the date the zap was made available. A zap will be either mandatory or optional, and it is either for a NEWDOS/80 module (i.e., one of the files on the NEWDOS/80 master system diskette) or for a non-NEWDOS/80 module. If it is mandatory zap to a NEWDOS/80 module, and your NEWDOS/80 system diskette is dated later than the zap, the zap will usually, but not always, already have been applied to your diskette.

Each zap will have a short explanation of the reason for it. Next will follow one or more zap areas, with each area composed of three parts:

10. The location on the diskette of the first byte of the area. This location will consist of 3 parameters and will be in the following format.

filespec1,relsector,relbyte

where

1. filespec1 gives the name or name/ext of the file to be zapped.
2. relsector is the relative sector within the file. relsector is in decimal.
3. relbyte is the relative location within the sector of the zap area's 1st byte. relbyte will be in hexadecimal but will not be suffixed with the character H.

Examples:

DIR/SYS,2,20
EDTASM/CMD,20,F6
YOURFILE,0,88

2. The old contents of the zap area. Each byte will be printed as two hexadecimal digits, and for readability the bytes will be separated by at least one space. If a hex digit position contains a - , then either Apparat doesn't care or doesn't know what exists in that hex digit before it is zapped.

3. The new contents to be zapped into the area, printed in the same format as for the old contents.

If a zap area covers more than 24 bytes, the format is changed so that both the before and after areas will be aligned to appear as the user will see them on the SUPERZAP display. This makes for easier viewing and zapping.

Many zaps really do not change the first and/or last bytes of the zap area. These bytes were included to help the user synchronize on the proper area, both before and after the zap, and to provide more verification bytes. However, it is not mandatory that the first and last bytes of the zap area be

used this way, and they usually won't be if the current zap area adjoins or overflows the area of another zap or if the zap area starts, ends, or overflows a sector boundary.

11.5. Zapping Procedure

To apply a zap, perform the following steps:

1. Make at least one backup copy of the diskette to be changed. NEVER, NEVER, NEVER apply a zap without first making a backup copy!!!
2. Execute DOS command SUPERZAP.
3. Mount the diskette containing the file to be zapped.
4. Enter the SUPERZAP function code DFS.
5. Enter the file's filespec, containing (1) the name or name/ext from the zap area location's 1st parameter (see section 11.4.1.1.) (if the file has been renamed, then use the applicable name/ext), (2) the access password, if required, and (3) the drive number.
6. Enter the zap area location's 2nd parameter (see section 11.4.1.2) as the relative sector number within the file.
7. The sector will be displayed to the user (see step 14 below). Find the zap area in the display, and verify that the old contents are as they should be. If they are not, then check if the zap you are about to apply is already applied; it may well be. If it is, then skip the current zap area and go on to the next. If it isn't, then check Apparatus.
8. When satisfied with the old contents, type MODxx without ENTER. xx is the zap area location's Ad parameter (see section 11.4.1.3.).
9. The cursor should appear over the first hex digit of relative byte xx. If the cursor does not appear, type in MODxx again. If the cursor appears over the wrong digit, check to make sure you are where you think you are. CAUTION!!! When the cursor appears, SUPERZAP is in modify (overwrite) mode; be careful what keys you press. In modify mode, left, right, up and down arrows and the space bar may be used to move the cursor.
10. To alter the hex digit in the cursor position, press the proper 0 - 9 or A - F key that represents the replacement value. The cursor will automatically advance to the next hex digit.
11. Type in all the new hex digit values.
12. If not satisfied with the changes, press Q to cancel the modification and return to the display.
13. When satisfied with the changes and ready to update them to the diskette, press ENTER. Then press Y, and when instructed, press ENTER again. SUPERZAP will exit modify mode back to display mode.

14. When in sector display mode (no cursor):

1. Press K if you wish to display another sector of the same file. Go to step 6.
2. Press J if you wish to go on to another file. Go to step 5.
3. Press X if you wish to return to the function menu.
4. Go to step 7 if there is another zap area for this same sector.

11.6. NEWDOS/80 Zap Distribution

Apparat requires registration of all NEWDOS/80 owners and will limit distribution of its zaps to registered owners. Please notice that, unlike other registration forms, the NEWDOS/80 registration card does not require the NEWDOS/80 owner to agree to anything; just let us know who you are!

Apparat does not guarantee that zaps will be distributed, as such distribution is a cost to Apparat over and above what the purchaser paid for NEWDOS/80. Apparat reserves the right to institute a charge for the zaps at some future time.

Zaps will be distributed by mail. Zaps will NOT be given over the phone. Distribution of zaps to all registered owners will occur whenever a large number of zaps has been accumulated. However, upon request, the latest zaps will be sent to individual registered owners, but please, if you are not having any trouble with your NEWDOS/80, don't ask.

When Apparat receives a registration card, the latest copy of the zaps will soon thereafter be mailed to the registered owner. This lets the owner know that Apparat has received the registration card and provides the owner with any zaps generated since either that manual (containing zaps as chapter 13) was made up or that NEWDOS/80 diskette was created.

11.7. Initial Installation of Zaps

When you first receive your NEWDOS/80, chapter 13 will contain the zaps outstanding at the time your manual was made up. Some of the pages for that chapter may have been inserted in the front of the manual at the last minute; find them and put them in chapter 13.

Next, make some backups of the NEWDOS/80 master diskette.

Now, since your NEWDOS/80 manual may or may not have been made up at the same time as your NEWDOS/80 diskette, you must synchronize the diskette with the zaps, if any, in chapter 13. Most of the mandatory zaps to NEWDOS/80 modules will already have been installed, but you must still check.

Using SUPERZAP, test if the highest numbered mandatory zap for a NEWDOS/80 module has already been installed. If it has, then you may assume all lower numbered mandatory zaps for NEWDOS/80 modules have been installed. This is not the case for optional zaps to NEWDOS/80 and any zaps to non-NEWDOS/80 programs. If this highest numbered mandatory NEWDOS/80 module zap has not been applied, then check the next lower numbered such zap until you reach one that has been installed. Then, from but not including that zap, start applying the higher numbered mandatory NEWDOS/80 module zaps in ascending numeric order. Higher numbered zaps may well zap over an area covered by a lower numbered zap.

Apparat has received many complaints from users who did not realize that some or all of these mandatory zaps were already applied to their diskette. As a general rule, but you must still check, a mandatory NEWDOS/80 module zap is installed on your diskette if your diskette is dated later than the zap.

As well as applying the mandatory NEWDOS/80 module zaps, you must apply the mandatory zaps, if any, to those non-NEWDOS/80 modules you are going to use with NEWDOS/80. You should also at least read the optional zaps so you know they exist.

Finally, though you will probably never know it, it is possible that your NEWDOS/80 diskette will have some mandatory zaps installed not yet listed in your chapter 13. This is not common, but such a thing has occurred. The zap sheets you receive in response to sending in your NEWDOS/80 registration card should cover those unknown but nevertheless already installed zaps.

11.8. Subsequent Installation of Zaps

When you receive a zap mailing from Apparat, you should apply the new mandatory zaps to NEWDOS/80 modules and to those non-NEWDOS/80 modules you are using with NEWDOS/80. Once again, you should at least read through the new optional zaps. There is no need to reread the zaps that you already have, as zaps are seldom updated and if they are, usually a subsequent zap refers to the change.

Remember, your NEWDOS/80 master diskette may already have some of the newer mandatory NEWDOS/80 module zaps applied; so check the highest numbered new zap and work your way down until you come to a zap that has been installed. Then start installing higher numbered zaps in ascending zap number order.

Never apply a higher numbered mandatory NEWDOS/80 module zap before applying all lower numbered mandatory NEWDOS/80 module zaps.

11.9. Diskette Update Service

In NEWDOS/80 version 1, due to the large number of zaps, Apparat instituted a NEWDOS/80 original diskette zap update service that is being continued for Version 2. This service does not replace the zaps but is intended for those users who would prefer Apparat to apply the zaps.

The user sends a package to Apparat containing his/her original NEWDOS/80 diskette, \$10.00 for service and handling, and a note explaining that the zap update is wanted. Address the package to:

APPARAT, INC.
NEWDOS80 Diskette Update Service
4401 S. Tamarac Parkway
Denver, Co 80237

Do not include any other information or requests in this package. Include in your note your phone number, your NEWDOS/80 registration number and the return address to be used.

Apparat will perform a full diskette COPY (without CBF option) from its then master onto your diskette, such that all NEWDOS/80 module mandatory zaps then outstanding will be included on your diskette. Your diskette will then be returned via UPS if possible (we can trace UPS better than the mail); otherwise, the mail will be used. Please, if possible, provide us with a street address.

The original diskette must still contain its original label with the registration number, which will be checked against your registration card.

The diskette must also contain the NEWDOS/80 system. If the registration number is missing or the diskette does not contain the system, the update will be denied. The \$10.00 service and handling charge applies each time an original NEWDOS/80 diskette is submitted and it must accompany the diskette. Be certain all non-NEWDOS/80 modules that you wish to keep have been taken off the diskette before sending it. If your original diskette is unchanged, then you have nothing to take off.

This zap update service includes the mandatory zaps to NEWDOS/80 modules only. It does not include optional zaps or zaps to non-NEWDOS/80 modules (i.e., SCRIPSIT, EDIT, etc.). This service does NOT include an upgrade to a new version of NEWDOS, if and when that occurs.

Do NOT send your diskette back to your dealer as dealers are not kept up to date on the current zaps. Send your diskette only to Apparat.

11.10. Zap Duplication.

All users keep many copies of NEWDOS/80, and single drive users are forced to have a NEWDOS/80 system on every diskette they use with NEWDOS/80. Once the new zaps have been installed correctly on one copy of NEWDOS/80 and these new zaps have been checked out, the user is now faced with the task of either zapping all the other diskettes or with copying the zapped files to those other diskettes. Through use of format 6 COPY (CBF) with the ILF and DFO parameters (the DFO parameters is defined below and not with COPY). Instead of specifying this procedure, the following example will be used instead.

Suppose that the modules SYS0/SYS, SYS2/SYS, SYS17/SYS, SYS14/SYS, BASIC/CMD, and DIRCHECK/CMD were changed by the latest zaps. The zaps were applied to one copy of NEWDOS/80, and NEWDOS/80 was then checked out

to make sure the zaps were OK. For the rest of this example, this diskette is referred to as the zapped diskette.

An ILF file (which is just like a chain file) is built containing the following records.

```
SYS0/SYS
SYS2/SYS
SYS17/SYS
SYS12/SYS
BASIC/CMD
DIRCHECK/CMD
```

This file is named ZAPNAMES/ILF and is placed on the zapped diskette. Next, a chain file is built containing one of the following two commands:

```
COPY,0,0,,NFMT,DFO,CBF,ILF=ZAPNAMES/ILF:0    single drive systems
```

or

```
COPY,0,1,,NFMT,DFO,CBF,ILF=ZAPNAMES/ILF:0    two drive systems
```

This file is named ZAPDUP/JCL and is stored on the zapped diskettes. Both of these files can be built using CHAINBLD (see section 6.6) or SCRIPSIT.

The zapped diskette will be considered both the SYSTEM and the SOURCE diskette and will be mounted on drive 0. The NEWDOS/80 diskette to receive the zapped modules will be considered the destination diskette, and, in the case of two drive systems, it will be mounted on drive 1.

Then, for every NEWDOS/80 diskette that is to receive the zapped modules, execute the DOS command:

```
DO,ZAPDUP
```

This DO command will cause execution of the COPY command contained in file ZAPDUP/JCL:0. Since the COPY command specifies an ILF file, only the files listed in that ILF file will be copied. Further, since the DFO option was specified, only those of the six files previously existing on both the destination and source diskettes are copied. For example, if DIRCHECK/CMD was not previously on the destination diskette, it is not copied to it.

Single drive system users will have to do a lot of diskette mounting. It is best to put a special marking on the zapped diskette to distinguish it from all the others.

Two drive system users will have only two responses per diskette copy.

Since the DFO (Destination Files only) option was not defined in COPY, it is defined here to mean that only files already existing on the both the destination and the source diskette are copied.

12. CONVERSION INFORMATION AND MISCELLANEOUS COMMENTS

This chapter contains Version 1 to Version 2 conversion information, miscellaneous information and changes to the information contained in other chapters as those chapters were already sent to the printers before the changes could be made.

12.1. RBAs gain in respectability

In late July, Apparat became aware that beginning with the Model III TRSDOS Version 1.3, TRSDOS is using RBA (Relative Byte Addressing) as the format for the EOF field in the directory FPDEs and for the EOF and NEXT fields in the FCBs. Finally, after 28 months, one of the major incompatibilities between NEWDOS and TRSDOS, that of the different handling of the FCB NEXT and EOF fields, will be mostly, if not fully, eliminated.

See section 5.7 for discussion of the FPDE EOF field in the 4th, 21st and 22nd bytes. See section 5.9 for discussion of the FCB EOF field in the 9th, 13th and 14th bytes and the FCB NEXT field in the 6th, 11th and 12th bytes.

See section 12.4 for NEWDOS/80 Version 2 incompatibility with Model I TRSDOS Version 2.3.

See section 12.5 for NEWDOS/80 Version 2 incompatibility with Model III TRSDOS Version 1.3.

TRSDOS's changing of the FPDE EOF field to RBA format is the correct move to make, but it has the unfortunate problem of making Model III TRSDOS 1.1 and 1.2 diskettes not directly readable on 1.3 and vice versa. Feeling that the 1.3 directory structure will become the Model III standard despite all complaints, the functions of the NEWDOS/80 COPY command (see section 2.14) that allow copying of files from and to Model III TRSDOS diskettes will work with the Model III TRSDOS 1.3 diskettes only.

When RBAs were instituted in March, 1979 as the NEWDOS format for the FCB NEXT and EOF fields, we also wanted to set the directory FPDE EOF fields to RBA format. Doing so would have made all NEWDOS diskettes incompatible with all existing TRSDOS diskettes and seriously reduced NEWDOS' usability. Since there are very few programs that actually read or write the directory FPDE EOF field and since the reason for changing to RBA formats is to eliminate confusing situations that could occur in FCB processing, Apparat decided to leave the directory FPDE EOF field alone. The procedure for converting from the FPDE EOF format used by NEWDOS and the old TRSDOSs to RBA format and vice versa is simple enough and doesn't cause confusion. The rules are:

To convert from the NEWDOS and old TRSDOS format to RBA format: if the lower order byte of the 3 byte value is non-zero, subtract 256 from the 3 byte value (or subtract 1 from the high order 2 byte value).

To convert from RBA format to the NEWDOS and old TRSDOS format: if the lower order byte of the 3 byte RBA value is non-zero, add 256 to the 3 Byte RBA value (or add 1 to the high order 2 byte value).

Even though at this time there are rumors of Model III compatible TRSDOS coming out for the Model I that will use the RBA format in the directory FPDE EOF field and even though Apparatus agrees that that field should be in RBA format, NEWDOS/80 for Version 2 will remain with the old format for that field.

12.2. Converting from Version 1 to Version 2 on the Model I

1. Most programs that worked on Model I NEWDOS/80 Version 1 will work on the Model I NEWDOS/80 Version 2.
2. The BREAK key enable/disable can no longer be controlled via bit 4 of 4369H. User program may continue to toggle this bit, but DOS ignores it. See section 2.8.
3. FCB changes (see section 5.9):
 1. Use of bit 2 (indicating track and sector operations) of FCB's 1st byte has been dropped.
 2. New definitions have been created for bit 3 of the FCB's 2nd byte and for bits 7 -5 of the FCB's 3rd byte.
 3. FCB's 17th through And bytes have been redefined.
4. Directory changes (see sections 5.6, 5.7 and 5.8):
 1. The GAT sector now accounts for lumps instead of tracks. Each byte within the 00 - BF range in the GAT now corresponds to a lump rather than a track, and granules per lump rather than granules per track is now used. The first byte of each extent element within FPDE's and FXDE's is now a lump number rather than a track number. The 3rd byte of the diskette's first sector (the boot sector) is now a lump number rather than a track number. Provided the proper GPL value is specified in PDRIVE, all Version 1 directories and boot sector 3rd bytes are directly usable on Version 2 and, with greater care, vice versa.
 2. Bits 7, 6 and 5 of the FPDE 2nd byte have been defined.
 3. The granule allocation table can now optionally use the first 192 bytes of the GAT sector. If the diskette's lump count is greater than 96 (60H), the granule allocation has overflowed into and negated the granule existence table (the lockout table).
5. DEBUG can no longer be enabled/disabled by the value in 4315H. User programs can continue to set this location, but DOS ignores it.
6. DEBUG can no longer be entered by pressing the BREAK key; only the 123 keys are used (see section 4.1).
7. PDRIVE has been greatly altered. Study section 2.37 carefully. The following PDRIVES must be used to read and write existing Version 1

diskettes on Version 2. These specifications must be used when making a diskette that will be read on Version 1.

1. PDRIVE,dn1,dn2,TI=A,TD=A,TC=35,SPT=10,TSR=3,GPL=2,DDSL=17,DDGA=2 is the specification for standard 5 inch, single density single sided diskettes. For 40, 77 or 80 track drives, set TC accordingly.
2. PDRIVE,dn1,dn2,TI=A,TD=C,TC=80,SPT=20,TSR=3,GPL=4,DDSL=17,DDGA=2 Use this PDRIVE setting for 5 inch, single density, double sided diskettes. For 35, 40 or 77 tracks, set TC accordingly.
3. PDRIVE,dn1,dn2,TI=BH,TD=B,TC=77,SPT=15,TSR=3,GPL=3,DDSL=17,DDGA=2 is the specification for 8 inch, single density, single sided diskettes used with the OMIKRON interface. Version 2 can handle up to SPT=17 for this type of diskette; you may want to covert your existing diskettes to gain the extra 12 percent space.
4. PDRIVE,dn1,dn2,TI=BH,TD=D,TC=77,SPT=30,TSR=3,GPL=6,DDSL=17,DDGA=2 is the specification for 8 inch, double sided, single density diskettes used with the OMIKRON interface. Version 2 can handle up to SPT=34 for this type of diskette; you may want to convert your existing diskettes to gain the extra 12 percent space.
5. PDRIVE,dn1,dn2,TI=CK,TD=E,TC=34,SPT=18,TSR=3,GPL=2,DDSL=17,DDGA=2 is the specification for 5 inch, single sided, double density diskettes with the PERCOM doubler interface. For 40, 77 and 80 track drives, set TC to 39, 76 and 79 respectively. If LNW interface, use TI=EK; if that doesn't work, try TI=CK.
6. NOTE!!! 5 inch, double sided, double density diskettes used on NEWDOS/80 Version 1 cannot be used on Version 2. The files on these diskettes must be moved, while using NEWDOS/80 Version 1, to either double sided, single density or single sided, double density diskettes, which can be used with Version 2. Once this is done, the file may be copied to a Version 2 double sided, double density diskette.
8. 5 inch double density diskettes are supported in Version 2 for the PERCOM and LNW double density modifications.
9. SYSTEM has been greatly expanded. Study section 2.46 carefully.
 1. Options AH and AK are dropped. Options AT through BN, except BL, have been added.
 2. Option BN decides whether NEWDOS/80 is to write single density directory sectors to be readable by Model I TRSDOS or readable by Model III NEWDOS/80. One or the other is allowed but not both.
 3. Option BJ allows NEWDOS/80 disk delay timing loops to be increased so that CPU speed up modifications can be active during disk I/O. NEWDOS/80 can handle most CPU speed-ups, but it cannot tolerate any slowdown of the CPU below the standard 1.772 megahertz speed.
10. COPY has been considerably changed. Study carefully section 2.14.

1. CBF will work even though the system diskette must be dismounted or if all three diskettes will use the same drive.
2. If you are using CBF (format 6) to copy the NEWDOS/80 Version 2 system to another diskette, then you MUST specify the FMT option. If you don't, the BOOT/SYS and DIR/SYS information may be wrong. If you are simply copying one or more of the system files to an existing system diskette (existing in the sense that it can already boot properly on the drive it is supposed to boot on) then you do not need to specify FMT. This information was not included in the CBF documentation and should have been.
3. COPY allows files to be copied back and forth between a NEWDOS/80 Version 2 diskette and a Model III TRSDOS Version 1.3 or higher diskette provided the proper PDRIVE setting is used (see PDRIVE TI flag M).

11. The DOS system ID formerly at location 403EH is now shifted to 4427H. In Version 1, 403EH contained either 80 (50H) or 128 (80H). In Version 2, location 4427H contains 130 (82H) identifying NEWDOS/80 Version 2, and location 442BH contains 01 if Model I and 03 if Model III.

12. None of the NEWDOS/80 Version 1 modules, including all the system modules, the BASIC modules and all other programs supplied on the master diskette, can be used with NEWDOS/80 Version 2. Therefore, the user files on Version 1 system diskettes must be copied to Version 2 system diskettes without copying any of the old Version 1 modules. For single drive users, this is a monumental task, but even multi-drive users must convert more than one system diskette. For each such system diskette, you may use the following procedure to copy your files.

1. Using a copy of the zap updated NEWDOS/80 master system diskette as both the system and source diskette, make another copy of that diskette using format 5 or format 6 COPY with the FMT option specified.
2. Kill off NEWDOS/80 Version 2 files that you do not want to keep. You could have effectively done this by using the ILF parameter in the above COPY, if that copy was format 6. Your ILF file can be built starting with the NWD80V2/ILF file provided on your NEWDOS/80 Version 2 master diskette and, using CHAINBLD/BAS or SCRIPSIT to delete lines for unwanted files. Remember to save the resulting file under a different name, which you will refer to in the ILF parameter of the COPY.
3. Using the resulting diskette again as the destination diskette and the old Version 1 diskette as the source diskette, perform a format 6 copy with the NFMT and the XLF=NWD80V2/XLF:0 parameters. This will copy all of your files from the Version 1 to the Version 2 diskette but will not copy any of the NEWDOS/80 Version 1 files, since they were all excluded by the XLF file. The file NWD80V2/XLF was included on the NEWDOS/80 Version 2 diskette exactly for this purpose and can be inspected via SCRIPSIT or CHAINBLD/BAS.

4. If you wish to copy the resulting Version 2 system diskette that now has your files as well back onto the old Version 1 diskette, you should do so using a format 5 or format 6 copy with the FMT option specified. This gets the Version 2 system and your files back onto the diskette with the old label.

12.3. Converting from Version 1 on the Model I to Version 2 on the Model III.

1. Most of section 12.2 applies here; read that section before reading this one. This section will deal only with Model III specifics.

2. Most user programs that were zapped to work with NEWDOS/80 Version 1 will work on the Model III NEWDOS/80 Version 2 with the following corrections:

1. All references to any bytes in the location range 4300H - 43FFH must be dropped or changed to different appropriate locations. This area is now the system sector buffer instead of the 4200H - 42FFH area used by Version 1.

2. The use of 4315H to toggle DEBUG must be dropped altogether.

3. The byte at 4312H used to enable/disable the BREAK key has been shifted to 4478H. The toggling of bit 4 of location 4369H must be dropped altogether.

4. The location of HIMEM has been shifted from 4049H - 404AR to 4411H - 4412H.

5. The location of the CLOCK has been shifted from 4041 - 4043H to 4217H - 4219H.

6. The location of the DATE has been shifted from 4044H - 4046H to 421AH - 421 CH.

7. The 25ms one byte cyclic counter has been shifted from 4040H to 441FH. The user timer interrupt routines still cycle based on 25ms increments even though the interrupts really occur every 1/30th or 1/125th of a second.

8. The 4410H vector used to insert a timer interrupt routine into NEWDOS/80's queue has been changed to 447BH (see section 3.8).

9. The DOS command buffer has been changed from starting at 4318H to start at 4225H.

3. The Model III NEWDOS/80 Version 2 diskette directories are in Model I NEWDOS/80 Version 2 format and are NOT compatible with Model III TRSDOS diskettes.

4. The Model III NEWDOS/80 Version 2 FCB format is the same as for the Model I NEWDOS/80 Version 2 and is NOT compatible with the Model III TRSDOS FCB format.

5. The following PDRIVE specifications must be used to read and write existing Version 1 diskettes on Model III Version 2. These specifications must be used when making a diskette that will be read on Version 1.

1. PDRIVE,dn1,dn2,TI=AK,TD=E,TC=39,SPT=18,TSR=3,GPL=2,DDSL=17,DDGA=2 is the specification for 5 inch, single sided, double density, 40 track diskettes. For 35, 77 or 80 tracks, set TC to 34, 76 and 79 respectively.

2. PDRIVE,dn1,dn2,TI=A,TDuA,TC=80,SPT=10,TSR=3,GPL=2,DDSL=17,DDGA=2 is the specification of a 5 inch, single sided, single density diskette. For 35, 40 or 77 track drives, set TC accordingly.

3. PDRIVE,dn1,dn2,TI=A,TD=C,TC=80,SPT=20,TSR=3,GPL=4,DDSL=17,DDGA=2 is the specification of a 5 inch, double sided, single density, 80 track diskette. For 35, 40 and 77 track drives, set TC accordingly

4. NOTE!!! 5 inch, double sided, double density diskettes used on NEWDOS/80 Version 1 cannot be used directly on the Model III. See section 12.2.7.6.

12.4. NEWDOS/80 Version 2 incompatibilities with Model I TRSDOS Version 2.3.

1. NEWDOS/80 maintains the NEXT field of the FCB in RBA format at all times. TRSDOS 2.3 maintains the NEXT field as an RBA whenever the lower order byte equals 0 or whenever the current write position is within a buffer that has been changed but not yet updated. In most other cases, TRSDOS tends to maintain the NEXT field equal to the RBA plus 256. At any one time, there is some confusion just what the NEXT field really means.

2. NEWDOS/80 maintains the EOF field of the FCB in RBA format at all times, and it updates the FCB EOF field for each byte written to the file, if indeed the EOF is to be changed. TRSDOS 2.3 updates the EOF only when the sector is actually written, though the low order byte is updated continuously during single byte or logical record writes. Thus if the current record would cause a change in EOF, EOF has two possible values, depending upon whether the current sector has pending data awaiting write or the current sector has already be written. Normally TRSDOS's FCB EOF value is an RBA value if the low order byte equals 0 and RBA plus 256 if the low order byte is non-zero.

3. Enabling or disabling of DEBUG in TRSDOS is still done by setting the byte at 4315H which is ignored in Model I NEWDOS/80 and must not be done in Model III NEWDOS/80.

4. Activation and deactivation of timer routines is done differently in the two systems (see sections 3.8 and 3.9 for the NEWDOS/80 methods).

5. Both Model I TRSDOS and NEWDOS/80 use essentially the same directory format except that TRSDOS is still limited to 35 track diskettes and a

two granule directory and that NEWDOS/80 uses some previously unused bytes and bits.

6. The following is a list of routines defined in chapter 3 that are common to both NEWDOS/80 Version 2 and Model I TRSDOS 2.3. Each routine performs nearly the same in both systems. The other chapter 3 routines are either not used in Model I TRSDOS or are defined for different functions. These common routines are:

0013H, 001BH, 402DH, 4030H, 4400H, 4405H, 4409H, 440DH, 441CH, 4420H,
4424H, 4428H, 442CH, 4430H, 4433H, 4436H, 4439H, 443CH, 443FH, 4442H,
4445H, 4448H, 4467H, 446AH, 446DH, 4470H, 4473H

12.5. NEWDOS/80 Version 2 incompatibilities with Model III TRSDOS Version 1.3

1. Model III TRSDOS diskettes are totally incompatible with NEWDOS/80 Version 2 diskettes. 5 inch, single density, single sided, 35 track diskettes with a two granule directory starting on lump 17 can be processed with Model III TRSDOS's convert program. Also, files can be copied back and forth between NEWDOS/80 Version 2 diskettes and Model III TRSDOS Version 1.3 or higher diskettes providing the PDRIVE specifications for the Model III TRSDOS diskette include the TI flag M.

2. Model III TRSDOS Version 1.3 has gone to using RBA values in the NEXT and EOF fields of the FCB and the EOF field of the directory. With this change to the FCB processing, NEWDOS/80 and TRSDOS has become more compatible than previously though, at this printing, just how close is not yet clear.

3. Model III TRSDOS uses a 50 byte FCB whereas NEWDOS/80 Version 2 stays with the old 32 byte format. NEWDOS/80 can use the 50 byte FCB area, but TRSDOS will clobber the 18 bytes following a 32 byte FCB. Users should study the specifications of the FCB's between the two systems as the differences are not detailed here.

4. The byte used to enable or disable the BREAK key is at 42AEH for Model III TRSDOS whereas it is as 4478H for Model III NEWDOS/80 and 4312H for Model I NEWDOS/80. If the byte equals 0C9H the BREAK key is enabled, and if the byte equals 0C3H the BREAK key is disabled.

5. The following is a list of the routines defined in chapter 3 that are common to both NEWDOS/80 Version 2 and Model III TRSDOS. Each routine performs nearly the same in both systems. The other chapter 3 routines are either not used in Model III TRSDOS or are defined for different functions. These common routines are:

0013H, 001BH, 402DH, 4030H, 4409H, 440DH, 441CH, 4420H, 4424H, 4428H,
442CH, 4430H, 4433H, 4436H, 4439H, 443FH, 4442H, 4445H, 4448H.

6. Refer to section 7.13 for comparison of the BASIC CMD functions offered in NEWDOS/80 with those offered for Model III TRSDOS.

7. Routing is handled somewhat differently in the two systems. Straightforward applications should be all right. DUAL is not implemented in NEWDOS/80.

12.6. Miscellaneous Comments

1. A very few users have coded system routines to be loaded by DOS' system routine loader, and these users should be aware that NEWDOS/80 Version 2 uses the system FPDE slots through SYS21/SYS. Whereas NEWDOS/21 and TRSDOS were limited to 14 system programs loadable by the system program loader NEWDOS/80 allows for 30 with FDE slot assignment continuing the same order established by the old TRSDOS. The code to activate a routine in one of these directory position dependent system modules is sent to the system in register A, must be greater than 1FH and in uuubbsss 8 bit format where:

sss+2 = the relative sector in the directory containing the FDE.

bb times 32 (20H) = the offset in the sector to the FDE.

uuu = a user defined code greater than 0.

A future release of NEWDOS will use system programs from SYS22/SYS and up; users should start from SYS29/SYS down.

2. All NEWDOS80 support programs use HIMEM high memory value in Model I locations 4049H-404AH (Model III locations 4411H-4412H) as upper memory limit.

3. (Model I only) During power on, reset or a jump to location control is passed to the ROM. To determine if the disk controller is present, the ROM tests the contents of location 37ECH, the disk controller status byte. If the value is either 00 or FFH, ROM assumes a non-disk system and proceeds to initialize non-disk level II BASIC. However, 00 is a valid disk controller state, meaning that the controller has no status and the drives are ready (the light is on). To avoid this unwanted entry into non-disk BASIC, wait until the ready light goes off before pressing reset.

4. To speed up disk operations when additional file space is allocated to a file, NEWDOS/80 allocates up to 4 granules at one time. There is a disadvantage to this, however. If two or more new files on the same diskette are open at the same time, it is quite possible to run out of file space, close all the files and then find out the diskette now has space, as CLOSE released the extra granules that files had allocated but not yet used.

5. NEWDOS/80 currently does not have any check on maximum-track number when it moves the diskette arm. If the track number exceeds the physical limits of the drive, the drive arm will bang against the stops for as many times as the track number exceeds the physical number of tracks for the drive. Since DOS retries I/O a number of times, it can be as long as one minute before the I/O is declared in error. To cut this interval short when this banging occurs, simply open the drive

door and wait till either the drives stop rotating or the error is declared. Then close the drive door.

6. The BASIC single stepping (CMD"F=SS") function does not allow time dependent functions such as an INKEY\$ loop to work. In the case of INKEY\$, if the user inputs a non-null key to INKEY\$ along with the ENTER that steps BASIC, the INKEY\$ key is ignored since it is seen before the ENTER. Also, the single stepping display does not work in 32 character display mode.

7. FORMAT correction. Parameter PFST is mutually exclusive with Y and with N.

8. COPY correction. If format 6 COPY (CBF) is used to copy the NEWDOS/80 system to a new system diskette, the parameter FMT must be specified in order that system files be allocated the required directory FPDEs, be assigned disk space in the required position relative to the directory, have the proper information placed into file BOOT/SYS. This type of COPY must be used whenever a system diskette is created whose PDRIVE specification is different from that of the source diskette.

INDEX

- A -		POPR	7-12
		POPS	7-12
		SASZ	7-12
		SS	7-14, 12-9
		SWAP	7-13
		I	7-10
		J	7-10
		L	7-10
		O	7-10, 7-14
		P	7-10
		R	7-10
		S	7-10
		T	7-10
		X	7-10
		Z	7-10
		doscmd	7-11
		COPY	2-9, 12-4, 12-9
		CREATE	2-18
		CVD	8-20
		CVI	8-20
		CVS	8-20
- B -		- D -	
BASIC MODULES	5-2	DATE	2-19, 3-11
BASIC2	2-5	DDGA	2-15
BAUD	2-44	DDND	2-12
BDU	2-13	DDSL	2-15
bit	10-1	DEBUG - 123	2-20, 4-1, 3-3, 12-2
BLINK	2-5	DEC	10-2
BOOT	2-6, 10-1	DFG - MINI-DOS	4-6
BOOT/SYS	5-1, 10-1	DFO	11-8
BREAK	2-6, 12-2	DI	7-4
buffer	10-1	DIR	2-20
byte	10-1	DIRCHECK	5-3, 6-12
- C -		directory	12-2, 10-2
		Directory Structure	5-4
CBF	2-14	DIR/SYS	5-1, 10-2
CHAIN	2-6, 4-7	DISASSEM	5-3, 6-5
CHAINBLD	5-3, 6-16	DISK BASIC	7-1, 8-1
chaining	10-1	activating	7-2
CHAJNTST	5-3	command truncation	7-4
character	10-1	direct commands	7-3
CHNON	2-7	enhancements	7-1
CFWO	2-14	I/O enhancements	8-1
CLEAR	2-8	file types	8-1
CLOAD	7-1	module overlays	7-1
CLOCK	2-9, 3-11	DO	2-22, 4-7
CLOSE	3-7, 10-2, A-9	DOS	10-2
CLS	2-9	DOS-CALL	4-12, 3-4, 10-2
CMD	7-8	DOS command (doscmd)	10-2
A	7-8	DOS ROUTINES	3-1
B	7-8	DOS SYSTEM MODULES	5-1
BREAK	7-1	DPDN	2-10
C	7-8	DU	7-4
D	7-9	DUMP	2-22
E	7-9		
F	7-9		
DELETE	7-13		
ERASE	7-12		
KEEP	7-12		
POPN	7-12		

- E -

EDTASM	5-3,6-14
EDIT direct commands	7-1,7-3
/ or shift up-arrow	7-3
; or shift down-arrow	7-3
.	7-3
,	7-3
:	7-3
@	7-3
up-arrow	7-3
down-arrow	7-3
EOF	10-3
EOL	10-3
EOM	10-3
EOR	10-3
EOS	10-3
ERROR	2-24,3-2
error messages	9-1,7-1
DOS	9-1,7-1
BASIC	9-2,7-2
extent element	10-3

- F -

fan	10-3
FCB	5-9,3-9,3-10,10-3
FDE	5-6,10-3
FF FILE	8-10,10-3,A-39,B-5,B-6,B-7
FI FILE	8-10,10-4,A-45,B-15
FIELD ITEM FILE	10-4
file	10-4
file item	10-4
filearea	10-4
filespec	10-4
FILE TYPE (ft)	8-10
FI	8-10,A-45
FF	8-10,A-39
MI	8-10,A-35
MF	8-10,A-30
MU	8-10,A-20
FILE POSITIONING (fp)	8-3,10-5,A-1
FIXED ITEM FILE	8-7,10-4
FMT	2-12
FORMAT	2-24,12-9,10-4
FORMS	2-26
FPDE	5-7,10-5
FREE	2-27
FXDE	5-9,10-5

- G -

GAT sector	5-5,12-2,10-5
GET	8-12,A-10
granule	10-5

- H -

hash code	10-5
hexadecimal	10-5

HIMEM	2-27,12-8,10-6
HIT sector	5-6,10-6

- I -

I/O error recovery	8-19
I/O link or path	10-6
ILF	2-14
IGEL	8-4,10-6
IGEL expression	8-5,10-6
IGELSN	10-6
item group	10-7

- J -

JKL	2-27,4-13
-----	-----------

- K -

KDD	2-13
KDN	2-13
KILL	2-28

- L -

LC	2-29
LCDVR	2-29
len	10-7
LIB	2-30
LINES	2-26
LIST	2-30
LMOFFSET	5-3,6-9
LOAD	2-31,3-7,7-4
V option	7-4
LOC	8-18,A-18
LOCK	2-3,2-40
LOF	A-17
logical record	10-7
Lower Case Suppression	7-8
LRECL	10-7
LRL	2-18
LSET	8-20
LUMP	12-2,10-7

- M -

MARKED ITEM FILE	8-7,10-7
MDBORT	2-31
MDCOPY	2-32
MDRET	2-32
MERGE	7-5
MF FILE	8-10,10-7,A-30,B-12,B-14
MI FILE	8-10,10-7,A-35, B-14,B-15,B-17
MINI-DOS - DFG	4-5
MKD\$	8-20
MKI\$	8-20
MKS\$	8-20
ms	10-7
MU FILE	8-10,10-7,A-20 ,B-2, B-3,B-4,B-9,B-10,B-11

- N -		REF	2-40
null	10-7		
null character	10-8		
null string	10-8		
NDNW	2-12		
NDN	2-13		
NDPW	2-12		
NFMT	2-12		
NOWAIT	2-44		
- O -			
ODN	2-1 2		
ODPW	2-14		
OPEN	8-9,3-5,3-6,9,10-8,A-6		
- P -			
PARITY	2-44		
partial record I/O	10-8		
PAUSE	2-33		
PDRIVE	2-33,12-2		
A	2-37		
DDGA	2-37		
DDSL	2-37		
GPL	2-37		
SPT	2-37		
TC	2-36		
TD	2-36		
TI	2-34		
TSR	2-37		
PFST	2-25		
PFTC	2-25		
PRINT	2-39		
print/input file	10-8		
PROT	2-3,2-40		
PSEUDO FIELD	8-17		
PURGE	2-41		
PUT	8-14,A-13		
- R -			
R	2-41		
RBA	12-1,10-8		
REC	2-18		
REF	7-7		
REGISTRATION	1-1		
REMB A	8-16,10-8		
REMRA	8-16,10-8		
RENAME	2-42		
RENEW	7-17		
RENUM	7-5		
Reporting errors	11-1,11-2		
reset/power-on	10-8		
ROUTE	2-42,12-8		
RSET	8-20		
RUN	7-4		
V option	7-4		
RUN-ONLY	7-2,7-8		
		sector	10-9
		SETCOM	2-44
		SN	2-13
		SOR	10-9
		SPDN	2-10
		SPW	2-12
		STMT	2-45
		SUPERZAP	5-3,6-1
		display mode	6-3
		function mode	6-1
		modify mode	6-4
		SCOPY	6-3
		SYSTEM	2-45,12-3
		AA	2-46
		AB	2-46
		AC	2-46
		AD	2-46
		AE	2-46
		AF	2-46
		AG	2-46
		AH	2-46
		AI	2-47
		AJ	2-47
		AK	2-47
		AL	2-47
		AM	2-47
		AN	2-47
		AO	2-47
		AP	2-47
		AQ	2-47
		AR	2-47
		AS	2-48
		AT	2-48
		AU	2-48
		AV	2-48
		AW	2-48
		AX	2-48
		AY	2-48
		AZ	2-48
		BA	2-48
		BB	2-48
		BC	2-49
		BD	2-49
		BE	2-49
		BF	2-49
		BG	2-49
		BH	2-49
		BI	2-49
		BJ	2-49
		BK	2-49
		BM	2-49
		BN	2-49
		SYSTEM Files Required	5-1
		SYSTEM reduced size	5-4

STOP 2-44

- T -

track 10-9
TIME 2-50
timer interrupts 3-3,3-4

- U -

UBB 2-13
UDF 2-4
UNLOCK 2-40
UPD 2-4,2-14
UPDATE SERVICE 11-6
USD 2-13
USR 2-14,2-41
user segmented file 10-9

- V -

VERIFY 2-51
vice 2-44

- W -

WIDTH 2-26
whole record I/O 10-9
WORD 2-44
WRDIRP 2-52

- X -

XLF 2-14

- Z -

ZAP 10-9
ZAPS
 Distribution 11-5
 Duplication 11-7
 Format 11-2
 Installation 1-4,11-5,11-6
 Procedure 11-4
 Update Service 11-6

- SYMBOLS -

/ext 2-14,2-41
*name routine 3-10,3-11
123 - DEBUG 2-19,4-1
/ or shift up-arrow 7-3
; or shift down-arrow 7-3
. 7-3
, 7-3
@ 7-3
up-arrow 7-3
down-arrow 7-3

APPENDICES

APPENDIX A

Understanding and learning to use the marked item and fixed item files specified in chapter 8 has proved difficult to the normal NEWDOS/80 user; therefore appendices A and B have been included to provide examples and more explanation in an effort to ease this difficulty. Nothing in appendix A or B is to be construed as overriding the specifications provided in chapter 8; the two appendices are provided simply and exclusively for examples and elaboration.

Appendix A was written by a user trying to cope with chapter 8 and is basically his understanding of marked item and fixed item files.

Appendix B is the NEWDOS/80 author's attempt to provide example programs of the 5 file sub-types: MF, MU, MI, FF and FI.

File Positioning

File Position (fp) is an operand in all NEWDOS/80 GETS and PUTS, and is specified in section 8.4.1. When omitted, a null operand is assumed. The fp operand otherwise commonly consists of a special character, occasionally followed by other special characters and/or expressions. One form of the fp operand consists of nothing more than a numeric expression. In the forms, which follow, special characters are to be used as shown. In those forms showing a prefixing special character adjoining some other character string, the special character does not necessarily have to be contiguous with the rest of the expression; it may be separated from it by a blank or space.

fp Value Meaning

(null)

If the file is an MU, MF or FF type file, and the REMRA is valid, the file is advanced to the next sequential record; in any other case, the current file position is not changed and processing continues from the position left at the termination of data transfer of the previous GET/PUT. Open leaves REMRA marked invalid for all file types, and sets current file position equal to 0 (except for mode "E", which causes current file position to be set equal to the FPDE's EOF value). The first sequential access for record segmented files always starts at current file position.

*

The current file position is not changed. This specification allows the continuation of processing of a particular record by a GET or PUT. It is primarily used to continue processing a record already partially read or written. For MU, MF and FF type files, it cannot be used to advance the file to the next sequential record, even though the file is actually already positioned at that record, having exhausted the bytes of the current record. To sequentially advance to the next record, use fp = (null).

#

If the REMRA is valid, the file is positioned to process that record again; an error condition is raised if the REMRA is invalid. For MU, MF and FF type files, this specification allows the reprocessing of the record currently being processed, from the beginning, perhaps with different variable names or expressions in the IGELs. For MI and FI type files, it allows the reprocessing of the same data item group as was processed by the immediately preceding GET/PUT.

\$

If the REMBA is valid, the file is positioned to begin processing at again it the point where the previous GET or PUT was at the end of its file positioning phase; an error condition is raised if the REMBA is invalid. This specification allows the reprocessing of a particular group of data by a GET/PUT, and is primarily used to reposition a file for partial record I/O. It functions in the same fashion for all NEWDOS/80 file types.

%

This specification performs a "pseudo FIELD" operation. No data transfer takes place; the filearea FCB is not changed; the file does not have to be open when this fp is used. It is used with FF and FI files to allocate user data strings of fixed sizes from the BASIC string storage area in high memory.

&

This specification is used only with PUTs, and has no effect on file positioning. It does however cause the current contents of a filearea buffer to be written to the diskette. It should be used whenever the data in the buffer is particularly sensitive. It may be used specifying the FAN of a PRINT file.

&&

This specification is similar to &, except that in addition the file's EOF is updated from the FCB to the FPDE. PUT fan,&& allows the programmer to force the EOF update to the FPDE without having to do a CLOSE.

!rba

Using this form of fp specification causes GET/PUT processing to begin at the specified location in the file where rba is a BASIC expression evaluating to a RBA value. For MU, MF and FF type files, the system checks to make sure that a record begins at the specified location. In the case of a MU file, the RBA value must point to an SOR item. This form of fp specification demands the greatest amount of care and premeditation on the programmer's part, as if it is used incorrectly, especially with FF and FI type files, it can be most disastrous. It is just about the only way to randomly access data stored in MI, MU and FI type files.

!%

This specification is basically the same as the !rba form except that the current EOF value is used as the RBA. It is commonly used to position a file for extension - that is, to add records/data to the end of the file. To extend a file it must be opened with mode "R"; mode "D" will yield an error if extension is attempted.

!\$rba

This specification allows the programmer to position the file for the next data transfer for that particular filearea,-without regard to the specific access technique or verb used for the transfer; no data transfer to user data areas occurs with this specification. No IGEL may be referred to or included in the GET/PUT using this specification. The positioning resulting from the use of this specification doesn't become effective until the next INPUT/PRINT or GET/PUT, and then only if no additional positioning is specified. It can be used to position a file for random access in a program which uses a subroutine containing a single GET/PUT having a (null) fp to do all file access; such a program could process sequential groups of records randomly distributed throughout a file.

!\$%

The basic function of this specification is identical to !\$rba, except that it uses the current EOF value as the RBA. The GET/PUT using this specification must not refer to or include an IGEL. Again, the file position resulting from this specification doesn't take effect until the next INPUT/PRINT operation, or the next GET/PUT (if another fp isn't specified).

!#rba

Used only with PUT, this specification sets the filearea's EOF value equal to the value rba. For the real EOF value of the file to be altered, that is, the one in the FPDE, the filearea must either be closed or a PUT && statement executed. The EOF value provided must be rational for the file type involved. For MF and FF files it must be an integral multiple of the file's standard record length.

rn (Record Number)

This specification is the same as the one supported by TRSDOS; rn is a numeric BASIC expression which evaluates to an integer value from 1 to 32767, inclusive. The specified record number is converted to an RBA which is then used in the same functional manner as !rba.

As mentioned above, certain forms for fp change REMBA, REMRA or EOF. For your convenience, the fp forms and their effects on these fields are summarized in the following decision table.

fp	REMBA	REMRA	EOF
(null)	1	1	6
*	1	2	6
#	3	4	6
\$	4	4	6
%	4	4	4
&	4	4	4
&&	4	4	4
!RBA	1	1	6
!%	1	1	6
!\$RBA	5	5	4
!\$%	5	5	4
!#RBA	4	4	1
RN	1	1	6

Meanings of codes in the matrix:

- 1 -- The field is set to the RBA resulting from that fp value.
- 2 -- If REMRA is invalid at the beginning of the statements execution, or it is an MI or FI file, the field is set to the RBA resulting from the fp value. In other words, it is set if the current file position is at the beginning of a record, otherwise it is unchanged.
- 3 -- The field is set equal to REMRA.
- 4 -- The field is not changed.
- 5 -- The field is set to an invalid value.
- 6 -- For output/update files, the field is changed if a PUT extends the file.

Altogether, there are four areas in an FCB relevant to 'file positioning'. These are:

Current File Position

This single field can be looked at as being 3 different values, depending upon where the GET/PUT is in its processing:

GPP1

The file position at the start of GET/PUT execution. Unless the file has been closed and re-opened, it is the same value left as GPP3 from the last GET/PUT for that filearea.

GPP2

The resulting RBA value after positioning has been done, and prior to any data transfer. GPP2 is the value saved as REMBA and REMRA whenever these values are set.

GPP3

The RBA value after the last byte of data transfer, if any, real or bypassed, has been accomplished.

REMRA

For MU, MF, FF and field item type files, it contains the RBA value of the beginning of the record in process. For MI, FT and INPUT/PRINT files, it is equal to REMBA. See GPP2 above.

REMBA

The RBA value where the previous data-transferring GET/PUT began its data transfer. If the file is record-segmented, and REMBA is at the start of a record, REMRA is set equal to REMBA. See GPP2 above.

EOF

The RBA value of the last byte of data in the file, plus 1. For MU, MF, FF and field item type files, it effectively points to the next sequential record to be written to the file. For MI, FI and INPUT/PRINT files, it effectively points to the next sequential byte to be written to the file.

The general method of managing the various fp values in the FCB goes as follows:

The file is moved from the current file position (GPP1) to the requested position, if necessary. This may include writing an updated buffer back to the diskette, computing the new sector address, and reading that sector into the buffer.

The RBA resulting from the requested positioning is placed in the current file position (GPP2).

REMBA is set equal to the current file position (GPP2).

If the file is an INPUT/PRINT file, is user-segmented, or is record-segmented and the current file position (GPP2) points to the start of a record, REMRA is set equal to the current file position (GPP2).

Data transfer, if any is requested, is done. The current file position (GPP3) contains the RBA of the byte following the last one transferred.

If the file has been extended, or the fp = INS, EOF is set to the appropriate value of the two.

OPEN

Any file must be opened before the data in it can be processed. The OPEN verb itself establishes an I/O link between the file and the applications program. The link's control information is maintained in the filearea (which contains a FCB). Once opened, the data in the file is made available to the program by means of INPUTS or GETS; data is placed on the file via PRINTS or PUTS. When the processing of the data is complete, the file should be closed, thus breaking the I/O link between the file and the program.

NEWDOS/80 supports five OPEN modes: "I" for sequential input (INPUT verb), "O" for sequential output (PRINT verb), "R" for random access input/output (GET or PUT verbs), "E" for sequential output starting at the current EOF for existing or new files (the "E" could be read as "extend"), and "D" for random access files which the user does not want expanded/lengthened with PUTS beyond the current EOF.

NEWDOS/80 BASIC marked item and fixed item file support allows the GET and PUT verbs to be used with all five modes. The general form of the NEWDOS/80 OPEN verb is:

1. OPEN m,fan,filespec
2. OPEN m,fan,filespec,lrec1
3. OPEN m,fan,filespec,ft
4. OPEN m,fan,filespec,ft,lrec1

where: m

is an expression evaluating to a string equal to "I", "O", "R", "E" or "D". It specifies the mode of access to be used for the file, as well as the initial positioning of the file.

fan

is the number of the filearea to be opened.

filespec

is an expression evaluating to the name of the file to be opened. The expression itself can be a string literal or constant.

ft

is an expression evaluating to a string equal to "FI", "FF", "MI", "MU" OR "MV". It identifies a particular NEWDOS/80 sub-file type, which will all be explained shortly. Whenever ft is used in an OPEN statement, GETS and PUTS are the only way to transfer data from and to the file. INPUTS and PRINTS must not be used. Neither may the BASIC FIELD statement be used. All GETs or PUTS used to transfer data must specify either an IGELSN or contain the IGEL itself. The applications program must not alter or directly reference the data in the filearea in any way. Two ft values require the specification of lrec1 in the OPEN statement; a third ft allows its optional specification.

lrec1

is an expression evaluating to an integer value between 1 and 256

for field item files and between 1 and 4095 for marked item and fixed item files. It must be specified for all record-segmented files (except field item files where 256 is assumed if `lrecl` is not specified), and specifies the exact length of all records in the file for field item, "FF" and "MF" files, or the optional maximum `lrecl` for file type "MU".

Note that the standard forms of BASIC OPEN have not changed (formats 1 and 2), thus allowing existing field-file and print/input file oriented applications to continue to function. The extensions to the standard forms identify the file as a NEWDOS/80 file, and define the file type and access technique used to retrieve and manipulate the data in it.

Of all the file types supported by NEWDOS/80, the easiest one to use and understand is "MU". It defines a file, which contains marked items, and is segmented into records of varying lengths. The length of a record is defined as the difference between the RBA of the record's SOR and the RBA of the next record's SOR or the RBA of the file's EOF, whichever follows. The record length need not be specified in the OPEN statement; but if it is provided, it specifies the maximum record length allowed in the file.

A record in an "MU" file can be updated with another record of the same or shorter length than it was originally created with, but it cannot be lengthened. When a record is updated with a record, which is shorter than the original record, the new record is padded on the right with fill items (bytes of hex '00') to the end of the original record. This shorter record can later be replaced with one, which is longer, as long as the new one is not longer than the record originally written to the file.

The "MU" file type is intended to replace BASIC's sequential input/output files accessed via INPUT/PRINT verbs. Its greatest strength is that no special delimiters have to be provided by the programmer to separate two contiguous string items (in BASIC sequential file support, a comma must be PRINTED between the strings for the INPUT to be able to separate them). A secondary benefit of "MU", and all other NEWDOS/80 BASIC files too, is that numeric values are stored on the diskette in their internal form. That is, for example, a double precision value is written as an 8-byte item, rather than an up-to-14 character item requiring conversion back to internal (8-byte) form on input. Don't forget that in the case of marked item files, such as "MU", a double precision item actually requires nine bytes due to the prefixing control character. If an `lrecl` is specified in the OPEN statement, it sets the maximum record length allowable for the file, and must allow for all control bytes (including SOR items) in each record.

The next most simple forms of file to use are "MF" and "FF". Both identify a file as record-segmented, and having records of fixed length. They both imply that all records have the same internal data structure, but do not guarantee that condition. The OPEN statement must specify the exact logical record length of all records in the file. In the case of "MF", the marking control bytes must also be accounted for in the length (note that an "MF" file doesn't use SOR items at the start of each logical record since BASIC knows where each record starts). Each GET/PUT checks the IGEL's data length against the `lrecl` specified at OPEN time, and raises an error condition if the IGEL's length is greater.

The most difficult forms of ft to use are "MI" and "PI". They specify that the file is record segmented entirely under user control. The "lrecl" must not be specified in the OPEN statement for these file types. These forms allow a file to contain a very complex data relationship, without BASIC's knowledge of the users data structure. That is, BASIC cannot advance from one user record to another.

CLOSE

The CLOSE verb breaks the I/O link set up by the OPEN verb between the BASIC application program and a file. Its general format has not been modified by NEWDOS/80.

Depending on the file's mode and type, the contents of the filearea buffer may be written to the diskette by this verb. For output and random-access files the file's directory entry is updated to reflect the current EOF value stored in the filearea's FCB.

GET

In field item (TRSDOS random) file processing, the GET statement is used to read a particular record into the filearea's buffer. The FIELD statement is then used to adjust the data pointers of string variables to address the buffer itself. This method of data access causes the file to be termed a field item file in NEWDOS/80 since all the other file types may also be used randomly.

In addition to continued support of field item files, NEWDOS/80's GET statement is used in marked item and fixed item file processing to transfer data from a file to user-specified variables, define the variables themselves, or position a file for later operations. The actual transfer of data from the diskette to the buffer occurs only as needed by the BASIC's determination of the IGEL data requirements in relation to the data currently in the buffer.

The general form of the GET statement is:

1. GET fan (a null fp is assumed)
2. GET fan,fp
3. GET fan,fp,igelsn
4. GET fan,fp,,igel

Formats 1 and 2 are used for field item files and are compatible with TRSDOS BASIC. They naturally may also be used in NEWDOS/80 BASIC application programs.

Formats 3 and 4 are unique to NEWDOS/80 BASIC. They must be used in data transfer GET whenever the filearea is open for marked item or fixed item file operations. Format 2's usefulness has been expanded by the addition of several new fp specifications unique to NEWDOS/80.

Format 3 specifies the location of the IGEL containing the data names, which are to contain the data at the completion of the GET; format 4 contains the IGEL as an integral part of the GET statement itself.

In NEWDOS/80, no function in the IGEL or the fp parameter may reference a filearea, even if that filearea is the same as that used by the GET or PUT statement.

At the successful completion of a GET statement, the filearea is left positioned at:

- a. the next byte of the file for fixed item files.
- b. the next item in the file for marked item files.
- c. the next 256 byte record for field item files.

If an error is encountered during the processing of a GET statement, the filearea is reset to its status and content prior to the execution of the statement. After correction of the error, the GET statement may be executed again. The contents of the variables named in the IGEL are entirely unpredictable when an error is detected, and should not be used unless the GET has been re-executed successfully.

When a GET statement refers to or contains an IGEL, successive file items are transferred to successive variables named in the IGEL.

For fixed item files:

String variables of the IGEL are filled with the number of bytes specified in the expression prefix. As a result, the length of the variable is made equal to the value of the prefix.

Numeric variables of the IGEL are filled with the number of bytes corresponding to that item's internal form. (Integer items are two bytes long, single precision items are four, and double precision are eight.)

Prior to the first GET which transfers data to user variables, a GET using `ft = %` may be issued. The file referenced by the `ft` need not necessarily be open when this GET is issued, as the purpose of this GET is to perform the pseudo FIELD function for fixed item file operations. As the IGEL items are processed, numeric variables are left unchanged, `(len)$` and `(len)#` items are ignored, and string variables have their length set to the value of the expression prefix, and are truncated or padded on the right with blanks as necessary. If a string variable exists at the time the pseudo FIELD is issued and its contents/value doesn't reside in the BASIC string area, its contents are moved there. This is done in an attempt to ensure that enough string space exists for continued operation, as the subsequent data transfer GETS will actually move data to the variable, rather than simply changing the variable's data pointer. Once referred to by a pseudo FIELD operation, string variables should have their contents changed only by LSET or RSET to ensure that the variable's lengths do not change. In NEWDOS/80 version 1, the pseudo FIELD function was required before any PUTS to a fixed item file; in version 2 this is not required and many programs using fixed item files will elect not to use the pseudo FIELD function at all.

If the file is record segmented and there are fewer bytes in the record from the current file position at the start of data transfer of the item than are requested by the IGEL item, a "RECORD OVERFLOW" error condition is raised.

For marked item files:

A null IGEL expression causes the corresponding file item to be skipped.

The expression prefix of a string variable is used to limit the number of characters actually transferred to the variable. If the file item is shorter than what the expression prefix allows, the length of the string variable is set to that of the file item. If the file item is longer than what the expression prefix allows, the file item is truncated on the right to that length, as would be done by an LSET.

SOR and fill items are skipped as they are encountered.

If the file item type and the IGEL item type are incompatible, a "TYPE MISMATCH" error is raised. If for example, the file item type were single precision and the IGEL item type were string, the error would be

raised. If however, the IGEL item type were integer, no error would be raised unless the file item's value exceeded what was legal for integer items.

If the file is record-segmented, and there are fewer items remaining in the record from the current file position at the start of data transfer of the item than are in the IGEL, a "RECORD OVERFLOW" error is raised.

Two special forms of fp may be used to set the file position for subsequent processing, regardless of the type of processing normally done for the file. These are fp = !\$rba and fp = !\$%. Use of either of these forms cause REMRA and REMBA to be marked invalid. Use of either of these ft values in a format 3 or 4 GET is invalid, as no actual data transfer takes place.

More than one GET may be used to retrieve successive file items from a single record. This technique is called partial record I/O. The first item in a record could, for example, identify the record as containing a name and address, a transaction number and amount, or an invoice number and expected ship date. The first byte could be read by itself and used to transfer control within the program to the appropriate routine to handle the data, which follows.

Partial record I/O as an access technique can be readily used with fixed item files and field item files. In field item files, the technique calls for reFIELDing when the new record is not the same type as the previous record. In marked item files, items to be bypassed in a record are simply left as null items in the GET's IGEL. In fixed item files, the length of the fields to be bypassed must be determined, and that sum be specified as the length prefix of a (len)\$ IGEL item, in order to position the record to the proper byte to be transferred. The real strength of partial-record I/O with fixed item files is that as little as a single field imbedded within a record can be updated independently of all other data in the record; with marked item files, all items beyond the one to be updated would first have to be read, then re-written with the item being updated to maintain their content. The primary benefit of partial record I/O is that several record formats can reside in a single file and only as much data need be transferred as necessary to identify the particular format.

PUT

In field item file processing, the programmer executes, if not done previously, a FIELD statement to define the variables' buffer overlaying main memory positions. Next, the values for those variables are moved into them using LSET or RSET statements. Lastly, the record is written (or buffered) using the PUT statement.

For marked item and fixed item file processing, the contents of BASIC variables are written (or buffered) using the PUT statement without the need of moving the data first to special encoded variables. Instead an IGEL is used to specify during the PUT which variables are to have their contents sent to the file.

Remember, no IGEL expression or the-fp expression may contain functions that reference a filearea.

The general form of the PUT statement is:

1. PUT fan (a null fp is assumed)
2. PUT fan,fp
3. PUT fan,fp,igelsn
4. PUT fan,fp,,igel

Formats 1 and 2 are used in field item file operations and are compatible with TRSDOS BASIC. They naturally may continue to be used in application programs running under control of NEWDOS/80.

Formats 3 and 4 are unique to NEWDOS/80 BASIC. One or the other or both must be used whenever data is transferred to the file during marked item or fixed item file processing. Format 3 specifies the location of the IGEL containing the expressions to be sent to the file; format 4 contains the IGEL itself as a part of the PUT statement. Format 2 PUTS may be interspersed with formats 3 or 4 to achieve the necessary file positioning for subsequent data transfer.

At the successful completion of a PUT statement, the filearea is left positioned at:

- a. the next byte of the file for fixed item files.
- b. the next item in the file for marked item files.
- c. the next 256-byte record for field item files.

If an error is encountered during the processing of a PUT statement, the filearea is reset to its status and positioning prior to the execution of the PUT statement. The data in the file as a result of the error is completely unpredictable, and will most likely cause errors on a subsequent GET. This situation occurs only during the updating of existing records; if possible and practical, a PUT should be issued later in an attempt to correct the error. In an effort to reduce the possibility of damage to the file when the file is opened using the "R" or "D" mode, NEWDOS/80 BASIC processes the IGEL twice in its entirety, once to catch errors in IGEL specification, and again to actually transfer the data to the buffer.

When a PUT statement refers to or contains an IGEL, the contents of successive IGEL expressions are transferred to the filearea buffer and become file items.

For fixed item files:

A string variable or expression may have a length different than the one allowed by the expression prefix in the IGEL. Strings which are shorter have the corresponding file item padded on the right with blanks; strings which are longer have the corresponding file item truncated on the right in the manner used by LSET. In other words, the expression prefix value determines exactly how many bytes are to be moved to the file item.

A record overflow error condition is raised if the logical record length is exceeded. During whole-record I/O, if the sum of all item lengths in the IGEL exceeds the LRECL, the error is raised. During partial-record I/O, if the sum of all item lengths in the IGEL exceeds the number of bytes left in the record, the error is raised.

Prior to the first PUT statement which actually transfers user data to the buffer, a PUT using `ft = %` may be issued. The file referred to by the `fan` need not necessarily be open at the time of this PUT, as its purpose is to perform the pseudo FIELD function. As the IGEL items are processed, numeric items are left unchanged, `(len)$` and `(lend` items are ignored, and string expressions have their length set equal to the value of the expression prefix, and are truncated or padded on the right with blanks as necessary. If the string variable exists at the time of the pseudo FIELD PUT and the string itself doesn't reside in the BASIC string space, it is moved there. Once referred to by a pseudo FIELD PUT statement, string variables should have their contents changed only by LSET or RSET statements to ensure that their lengths do not change. In NEWDOS/81 version 1, this pseudo FIELD function was required before any PUTS to a fixed item file; in version 2 this is no longer required and many programs using writing to fixed item files will elect not to use the pseudo FIELD function at all. The pseudo FIELD function is left in existence for the programmer who wants to assure IGEL related string variables maintain the required length at all times.

For marked item files:

SOR and fill items are inserted into the filearea buffer as dictated by the file's `ft`, the PUT's `fp` and the IGEL data length versus the file's record length.

Nearly anything syntactically legal on the right hand side of a LET expression's equal sign is legal as an expression in an IGEL referenced by a PUT statement, excepting that a filearea may not be referenced in such an expression. Specifically excluded from appearing in any IGEL expression are LOC, LOF, EOF and any other expression, which references a `fan`.

When a string expression in an IGEL has a length prefix, the prefix determines the maximum number of characters to be written to the file. If the string is shorter than the expression prefix allows, the string

is written to file as is. If the string is longer, the corresponding file item it is truncated on the right as would be done by an LSET operation.

Strings require either one or two marking bytes, depending on the number of bytes in the string. If the string has from 0 to 127 bytes in it, it requires only one marking byte to describe it on file. If it has 128 bytes or more, then two marking bytes are required to describe it. All these marking bytes must be allowed for when specifying an lrecl at open time.

Numeric IGEL expressions are placed in the buffer in their internal BASIC form: 2 bytes for integers, 4 bytes for single-precision numbers and 8 bytes for double-precision numbers. Don't forget that each individual file item has a marking byte associated with it, and that the correct lengths of the item types just mentioned are 3, 5 and 9 bytes.

Numeric literals and expressions in the IGEL are first converted to the most compact internal BASIC data type that preserves their precision before being sent to the file. For example, the numeric literal 3.14159 would be sent to file as a single precision number (5 bytes including marking byte); the value resulting from LEN(A\$) minus LEN(B\$) would be sent to file as an integer number (3 bytes including marking byte).

Two or more PUT statements may be used to output all the items of a record. The number of bytes actually comprising a single logical record cannot exceed the lrecl value specified in the OPEN statement, or the system maximum of 4095 bytes.. Any attempt to exceed either of these limits results in a "RECORD OVERFLOW" error.

In the case of MU and MF type files opened for random access updating purposes, the record existing on file, from the current file position at the beginning of data transfer for the PUT, to the record's end (defined by the next SOR, or EOF) is replaced in its entirety. If the cumulative IGEL data length is less than the file record's remaining length, the IGEL data is sent to the file and padded out with fill items to completely fill the file record. Be very careful when operating in this mode, because if the PUT's IGEL defines fewer items than exist in the file record at the time of update, the excess file items are eliminated; later GET statements will encounter problems if they expect the original number of items to be present in that record.

Items in a MI type file cannot be updated as the system has no idea where the user's record ends, and therefore cannot pad to the end of the record as it does for MU and MF files.

For both fixed item and marked item files:

The filearea's buffer is actually written to the diskette when:

The last byte of the buffer is filled with data from the IGEL, and more data has yet to be moved.

A PUT statement with an fp of "&" or "&&" is executed, causing the buffer to be written to the diskette in its current state.

The file is closed, explicitly by fan, or implicitly by a general (non-specific) CLOSE.

If the data in the file be especially critical, the programmer should consider the use of PUT statements with the fp of "&". This will cause the filearea's buffer to be written to the diskette without disturbing the current file positioning. If there is no data in the buffer waiting to be written to the diskette, this particular PUT statement will be ignored. Should some other filearea used by the program require the data in this filearea to be disk-resident, the fp of "&" must be used. Don't overlook the fact that an fp of "&" is used only in a format 2 PUT; any data to be written to file must first have been placed there by a format 3 or 4 PUT. The use of fp = & is not restricted to marked item or fixed item files - it may be used with field item files or print/input disk files also.

Everything said above for the PUT fan,& statement also applies to the PUT fan,&& statement which, in addition, writes the file EOF from the filearea's control information back to the file's directory entry.

Two special forms of fp may be used to set the file position for subsequent processing normally done for the file, regardless of the actual type of processing involved: GET, PUT, INPUT or PRINT. These are fp = !\$rba and FP = !\$%. Use of either of these forms causes REMRA and REMBA to be marked invalid. The file is positioned so that the next GET/PUT/INPUT/PRINT verb begins processing either at rba or EOF, if no further fp is specified. No data movement occurs using these fp values, as they are allowed only in a format 2 PUT.

A PUT statement using an fp of !#rba causes the file's EOF to be set to the RBA value rba. Don't forget that the EOF value is not written to the file's FPDE until a CLOSE or a PUT fan,&& statement is executed. The EOF may be changed many times in this fashion before it is made final. An error condition is raised if the OPEN statement's mode was "D", and the RBA exceeds the current EOF value. This fp value may only be used in a format 2 PUT.

As was the case with GET for sequential input, the PUT statement can be used in a sequential output mode. A marked item or fixed item file can be created sequentially with PUT statements after having been opened with mode "O", and later read sequentially with GETS after having been opened with mode "I". The same file can be updated randomly by use of GET and PUT statements when the open mode is "R" or "D". Single data fields in FF and FI type files can be updated using partial record I/O access techniques.

Should a particular data file be especially sensitive, and require read-only random access, the use of open mode "R" is not required; open mode "I" may be used instead. The use of this particular mode will cause any PUT attempted to get a "BAD FILE MODE" error.

LOF

The function of the LOF statement is to return to the programmer the record number of the last record of the file. Its general format is:

LOF(fan)

The fan specifies the number of the filearea for which the last record number is being requested. If the file is empty, a zero is returned. LOF naturally may be used only with field item, MY and FF type files.

LOC

The LOC function, in TRSDOS BASIC, returned to the programmer, the record number last accessed via GET/PUT for a specified filearea. In NEWDOS/80 BASIC, its function has been expanded to allow the programmer to find the file location of a group of items, records or the files' EOF, or determine if the current file position is exactly at or beyond the file's EOF. Its general formats are as follows:

1. LOC(fan) performs essentially the same as in TRSDOS
2. LOC(fan)\$
3. LOC(fan)%
4. LOC(fan)!
5. LOC(fan)#

where fan specifies the filearea number containing the requested information.

Format 1 (no suffix) is the one used in TRSDOS BASIC. For field item files (as are supported by that BASIC) and MF and FF files, it returns the number of the record most recently read or written via GET/PUT. If the file has not been accessed, a value of zero is returned, except in the case of a file opened using mode "E", where the record number of the last record in the file is returned. If the file being referenced is not made of fixed-length records, a "BAD FILE MODE" error condition is raised.

Format 2 (" \$" suffix) is used to provide a true/false indication of the relationship of the filearea's positioning to the file's EOF. It returns a -1 (BASIC IF statement 'true') or a 0 (BASIC IF statement 'false') as follows:

For record-segmented (fixed item, MU, MF and FF type) files:

If the REMRA is valid, and the RBA of the start of the next record (not necessarily the current file position!) is equal to or greater than the EOF value, a 'true' value is returned; otherwise a 'false' value is returned.

If the REMRA is invalid and the RBA of the current file position is equal to or greater than the EOF value, a 'true' value is returned; otherwise, a 'false' value is returned.

For user-segmented (MI and FI type) files, and for print/input files:

If the RBA of the current file position is equal to or greater than the EOF value, a 'true' value is returned; otherwise, a 'false' value is returned.

Format 3 ("% " suffix) returns to the programmer the file location of the current file EOF in RBA format. This value can be used in the development of indices to the file, where the indexing item is built prior to the data record being added to the file at the EOF location. Using this form of LOC allows indices to be created during the sequential creation of the prime data file.

Format 4 ("!" suffix) returns the RBA value of the next logical record for field item, MU, MF and FF type files, if the REMRA is valid. In all other cases (including print/input files), it returns the RBA value of the current file position. For record segmented files, the value returned can be used to create an indexing item for the sequential record before the data record has been written to the file. For user-segmented and print/input files, the value returned can be used to create an indexing item for the group of data items prior to writing them to the file. For the indexing value to really be good, a PUT with a null fp, or a PRINT, must be used to write the data; nearly all other fp forms will cause the RBA value returned to be different from the actual location of the data. As with format 3, this form can be used to create indices as a sequential file is being written.

Format 5 ("#" suffix) returns the current REMRA in RBA format. A "BAD FILE MODE" error condition is raised if the REMRA is invalid, due for example, to the use of an FP m !\$%. This too can be used for all file types to create indexing items for records or groups of data after, however, the record or data group has been written.

By using the values returned by LOC(fan)%, LOC(fan)! or LOC(fan)#, the programmer is able to build indices to either records (record-segmented files) or groups of items (user-segmented files and print/input files). The values, returned can be included in records/file items and later used to position the filearea via fp types !rba or !\$rba.

MU FILES

The MU file type is the easiest of all NEWDOS/80's file types to implement. When it was originally conceived, it was intended as a replacement for TRSDOS's sequential file support. In TRSDOS, sequential files could not be updated; in NEWDOS/80 all but print/input and MI type files can be updated.

The MU type file is segmented into records of varying lengths and each record is detectable by the system. This attribute relieves the programmer of the need to be aware of the size of each record. The programmer can impose a smaller record size maximum than the system's maximum of 4095 bytes by specifying a lrecl value in the OPEN statement. Any record exceeding the maximum record length will cause a "RECORD OVERFLOW" error condition.

Besides being record-segmented, the file items in a MU file are all marked. The marking bytes occupy space on the file, and must be included in any record length calculations along with the SOR byte, which marks the beginning of each record. These marking bytes identify the type of data, which follows the byte, and in the case of strings, tells the system the length of the string. Strings may be 0 to 255 bytes long, just as in BASIC; strings of 128 to 255 bytes require 2 marking bytes instead of the 1 required by all other items. Numeric items are stored on the disk in their internal form: integers as 2 bytes, single-precision items as 4 bytes, and double-precision items as 8 bytes. Don't forget that as marked-file items these lengths must be increased by 1 to 3, 5, and 9 bytes respectively.

Even though the numeric items are stored in their internal forms on the disk in all the NEWDOS/80 file types, BASIC's CVx and MKx do not (indeed, must not) be used to perform a pseudo-string conversion in order to cause this form of data storage to occur; CVx and MKx must still be used to accomplish this form of data storage for field item files, as was the case with TRSDOS BASIC.

A MU file can be created by specifying "0" as the mode in the OPEN statement; the file will be created using the data in successive PUTS without regard to the file's existence at the time of the open. A MU file may also be created using mode "R" in the OPEN statement only if the file did not exist prior to the open. A third method of creating a MU file is to use mode "E" in the OPEN statement for a previously non-existent file, or an existing file, which is empty.

A existing MU file can be expanded sequentially by specifying mode "E" in the OPEN statement. As noted above, if the file is empty, it will effectively be created rather than expanded/extended. An alternate method of sequentially expanding a MU file is to specify mode "R" in the OPEN statement. In this mode if non-null fp's are specified, the system writes padding bytes from the current EOF to the specified beginning of the new record. Any PUT to a file position less than the EOF causes an updating action to occur, not an extension of the file.

A MU file may be accessed sequentially by specifying "I" as the mode in the OPEN statement; use of this mode prevents accidental updates from occurring. The file may also be accessed randomly when opened with mode "I". If the file is non-existent at the time of the open, an error condition is raised. A MU file may also be accessed sequentially by specifying "R" or "D" as the mode

in the OPEN statement. Using these modes, if the file was non-existent prior to the open, any GET issued without a prior PUT and subsequent repositioning will cause an error condition to be raised.

A MU file may be updated by specifying mode "R" or mode "D" in the OPEN statement. The use of mode "D" precludes the expansion of the file. In either of these modes, anything from an entire record to a single item may be updated, depending upon the fp values used and the contents of the IGEL.

To understand the workings of the system on a MU type file, we'll do the following things. First, we'll create a MU file using a very simple, short BASIC program. Then, by working in the so-called calculator mode, we'll access the file and update it. To create the file, enter and RUN the following BASIC program:

```
10 CLEAR 250
20 OPEN "O", 1, "MU/DAT", "MU"
30 PUT 1,,,"ABCDEF","2ND STRING";
40 PUT 1,,,STRING$(120,"*")+ "0123456789";
50 I%=2:I!=4:I#=8
60 PUT 1,,,I$,I%,I!,I#;
70 CLOSE
```

Save the program with an appropriate name just in case you need it later.

Now, notice that the program uses the simplest form of IGEL in statements 30 and 40; the values to be written to the file are in the IGEL proper. The PUT at 60 references the four different BASIC data types: string, integer, single precision and double precision. Notice also that no lrec1 specification was in the OPEN statement. This allows the records to be as much as 4095 bytes long.

Run the program to create the file named "MU/DAT". For study purposes, run the SUPERZAP program using DFS to read the sector written by MUFIL.

The first byte of the sector is a hex 70. This is the SOR byte. All records in a MU file start with this byte. Be aware that not all hex 70's are start of record bytes, however, that particular bit configuration can occur in numeric values as well as in strings where it is a lower-case "p".

The second byte is a marking byte identifying the next 6 bytes as a string. Adding 6 to the displacement of the first byte of the string will give us the displacement of the marking byte for the second string (a hex 8A). It defines a string 10 bytes long. If you now count to the 11th byte down from that marking byte, the SOR byte for the second record will be found (at displacement hex 13). The following marking byte (a hex 71) identifies a string of greater than 127 bytes long; the byte following that marking byte contains the length, and is not a part of the string data itself. A little hex arithmetic at this point will show that the SOR byte of the third record will be found at displacement hex 98. The marking byte following that SOR identifies a string zero bytes long: a null string. The next marking byte (hex 72) identifies the following 2 bytes as an integer number. Following the integer is a marking byte (hex 73) identifying the next 4 bytes as a single precision number. Following that number is a marking byte (hex 74) identifying the next 8 bytes as a double precision number. At this point

(displacement hex AB) we've exhausted the data we actually wrote to the disk; any data, which follows, is unpredictable.

Now that we've seen how data is stored in a MU file, as well as any other marked item file for that matter, we'll access the data using GETS in the "calculator mode" and analyze the results. Later, we'll introduce a few errors. Before going any further, return to the BASIC READY state, enter CLEAR 50 and NEW, and type in the following three-line program (this will save steps later).

```
10 PRINT LOC(1)$; "$ EOF TEST "; LOC(1)%; "% EOF RBA"
20 PRINT LOC(1)!; "! NEXT RCD RBA ";
30 IF LOC(1)! = 0 THEN PRINT ELSE PRINT LOC(1)#; "# REMRA"
```

The purpose of the program is to display the file positioning values available to us. For the sake of clarity, the first character of the string identifies the LOC suffix used to get the value displayed and the remainder of the string a mnemonic associated with that particular LOC function. You may want to save this program also, as it will be used in experiments with all the other file types later.

The first thing to do now is to open the file for input. Type in:

```
OPEN "I", 1, "MU/DAT", "MU"
```

Now enter "GOTO 10" to run the program entered a moment ago. (You must use GOTO rather than RUN because RUN closes any open files.) The system will respond with:

```
0 $EOF TEST 171 % EOF RBA
0 ! NEXT RCD RBA
```

Notice that the REMRA value isn't printed. That's because the value hasn't been set yet, and is marked as invalid by the system. Because the program we entered isn't too smart, it simply checks for a zero next record value, rather than attempting to be sensitive to the actual validity of the REMRA.

Now we'll read the first record in its entirety. Type in:

```
GET 1,,,A$,B$; : PRINT A$, B$ : GOTO 10
```

The system will respond with:

```
ABCDEF 2ND STRING
0 $ EOF TEST 171 % EOF RBA
19 ! NEXT RCD RBA 0 # REMRA
```

Notice that the two EOF related values have not changed, but that the next record RBA has. It now contains the decimal displacement of the SOR byte of the second record. This is the normal action of the GET on a record-segmented file. Notice also that the REMRA has now appeared, and that it has a value of zero. Remember that for record segmented files the REMRA contains the RBA of the latest record involved in the GET or PUT for that filearea, unless its has been marked invalid due to the use of OPEN or !\$RBA.

Now we'll go back and read the first record again in its entirety by using the fp value which causes file positioning back to the REMRA value. To prove the record has been read a second time, we'll reverse the order of the variable names. Type in:

```
GET 1,#,,B$,A$; : PRINT A$, B$ : GOTO 10
```

The system will respond with:

```
2ND STRING   ABCDEF
0 $ EOF TEST   171 % EOF RBA
19 ! NEXT RCD RBA   0 # REMRA
```

Again, the EOF values have not changed. This time, however, neither have the other two values. This is because the file's next record pointer was changed to the REMRA value prior to the data transfer. The next record pointer was then moved to the REMRA, followed by the transfer of the data to the named variables. The same general method is followed when !rba is specified for the fp.

Let's get daring now, and ignore the contents of the next record (the one with the 120 asterisks in it), and at the same time position ourselves to process the third record. Type in:

```
GET 1,,,; : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST   171 % EOF RBA
152 ! NEXT RCD RBA   19 # REMRA
```

Nothing really surprising there; again, in the case where no file positioning was specified in the GET, the next record RBA was moved to the REMRA. With the lack of variable names in the IGEL, no data transfer occurred, and the file was left positioned to the record's first item.

Now let's try some of partial record I/O. We'll start by transferring only the string from the third record, and leave ourselves positioned so that the next transfer will begin at the integer. Type in:

```
GET 1,,,A$; : PRINT A$ : GOTO 10
```

The system will respond with:

```
(blank line)
-1 $ EOF TEST   171 % EOF RBA
171 ! NEXT RCD RBA   152 # REMRA
```

Several things should be noted here. Since the file was left positioned to the 2nd record's 1st item by the previous GET and the GET in this example specified fp = (null), the file was automatically advanced to the beginning of the third record's 1st item by this GET's file positioning phase. Then the 3rd record's first item was read, and the file was left positioned to the 3rd record's second item. We've started processing the last record in the file. The system hasn't told us that, but has made the information available to us through the LOC(fan)\$ statement. in common sequential data processing

situations, the EOF status of a file is tested as a function of the GET logic, and transfer of control is made to an end-of-data routine specified by the programmer. As no provision has been made for the specification of such a routine in NEWDOS/80, the EOF status of the file must be tested immediately prior to the GET statement attempting to transfer the next record's contents into memory, and appropriate action taken if the EOF condition is found to be true.

Notice that the LOC(fan)! value is the same as the EOF RBA value, even though we transferred only the first item of the last record. This is because in the case of record-segmented files, the function returns the RBA of the next record. Only when it is used on a user-segmented file does it return current file position. If you've gone back to chapter 8, you've seen that there's no way to get the current file position back from the system. There isn't, nor is there a way to get the REMBA either. Somebody out there will probably find a way via PEEKS and so on, but the fact remains that BASIC itself doesn't have provision for telling you simply and directly.

To show that we are indeed positioned at the record's 2nd item, the integer, we'll read just that field. Type in:

```
GET 1,*,,I; : PRINT I : GOTO 10
```

The system will respond with:

```
2
-1 EOF TEST    171 EOF RBA
171 NEXT RCD RBA  152 REMBA
```

Did you notice the variable type of "I"? It's single precision, but the file item transferred to it was an integer. The changing of type between a file item and a variable is allowed, so long as it is allowed in BASIC.

Now let's go back and transfer the integer and the single precision items using the REMBA to position the file before the transfer. Type in:

```
GET 1,$,,K,J; : PRINT J; K : GOTO 10
```

The system will respond with:

```
4 2
-1 $ EOF TEST    171 % EOF RBA
171 ! NEXT RCD RBA  152 # REMBA
```

The REMBA was set to the file's RBA at the start of the previous GET. Regardless of the number of fields transferred or bypassed, the starting byte RBA is remembered. Again, none of the LOC functions has changed.

To prove that the REMBA hasn't changed with the multiple file item transfer, let's transfer the integer and the double precision items next. Type in:

```
GET 1,$,,J,,I; : PRINT I; J : GOTO 10
```

The system will respond with:

```

8 2
-1 $ EOF TEST 171 % EOF RBA
171 ! NEXT RCD RBA 152 # REMRA

```

Notice that by omitting a variable name in the IGEL in the position where the single precision file item occurs, that the file item is bypassed. Again, both file items have their types changed as they are moved to the variables.

Now we'll try some RBA positioning to see how that works. Type in:

```
GET 1,10,,A$; : PRINT A$ : GOTO 10
```

The system will respond with:

```

ABCDEF
0 $ EOF TEST 171 % EOF RBA
19 ! NEXT RCD RBA 0 # REMRA

```

The use of a specific RBA provided by the programmer, whether it's a number as in this example, or some variables contents, or an expression, causes the RBA to be moved to the next record pointer just as the REMRA is moved there when "#" is used for fp. The sequence of actions is the same from that point on for the two fp's just mentioned.

Let's try the other RBA positioning technique. Type in:

```
I=152 : GET 1,!$I : GOTO 10
```

The system will respond with:

```

0 $ EOF TEST 171 % EOF RBA
152 ! NEXT RCD RBA
BAD FILE MODE

```

Hey! Was that supposed to happen? You bet! Both the REMRA and REMBA were tagged as invalid by the system due to the fp type used. It does nothing more than set the next record pointer. No data transfer occurs.

Now we'll try it again, but this time with a "later" data transfer. Type in:

```
GET 1,!$19 : GET 1,,,A$ : PRINT A$ : GOTO 10
```

The system will respond with:

```
OUT OF STRING SPACE
```

Another error? Why? Because when we started this session, we did a CLEAR 50, and the string we're trying to transfer is 130 bytes long. Don't forget that NEWDOS/80 doesn't change the string variable's pointer to point to the buffer, but moves the string to the BASIC string space at the top of memory as if a LET statement had been executed. Now type in:

```
GET 1,,,(10)A$; : PRINT A$ : GOTO 10
```

The system will respond with:

```
*****
0 $ EOF TEST    171 % EOF RBA
152 ! NEXT RCD RBA  19 # REMRA
```

NOTE: the same file item was inputted as for the previous GET. Due to the error that occurred, the filearea, but not the data, was restored to what it was at the beginning of that previous GET. Note that only the first 10 asterisks of the 120 in the file item were transferred to A\$.

That just about exhausts the fp's we can use. The ones not covered yet are fairly well explained in chapter 8. It is time now to try some updating of records, both in whole and in part. Before we can do that however, the file must be opened for input and output. Type in:

```
CLOSE : OPEN "R", 1, "MU/DAT", "MU" : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST    171 $ EOF RBA
0 ! NEXT RCD RBA
```

That is just as it was after the open for input only. The mode we just specified allows the file to be expanded (which we will do shortly). If we wanted to not allow the ability to expand the file beyond its existing EOF, we would have specified mode "D".

First, let's simply replace the first record on the file with a single field. Type in:

```
PUT 1,,,"RECORD REPLACED"; : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST    171 % EOF RBA
19 ! NEXT RCD RBA  0 # REMRA
```

Notice that the next record pointer is pointing to the second record, just as if a GET were issued.

Now, let's replace the double precision value in the third record with 3 times its complement. Type in:

```
I=152 : GET 1,!I,,,";
GET 1,*,,D#; : PUT 1,$,,3*-D#; : GOTO 10
```

The system will respond with:

```
-1 $ EOF TEST    171 % EOF RBA
171 ! NEXT RCD RBA  152 # REMRA
```

Again, the system is ready to process the next record even though it's positioned at EOF. We can't transfer any information from this file position, but can write additional new records to the file.

To demonstrate this, type in:

```
PUT 1,,,"THIS IS THE FOURTH RECORD"; : GOTO 10
```

The system will respond with:

```
-1 $ EOF TEST    198 % EOF RBA
198 ! NEXT RCD RBA    171 # REMRA
```

It's easy to see that the file has been extended. You should be aware that the new EOF hasn't yet been recorded in the FPDE in the directory. If there were to be a power outage at this point, our little example file would show no change from when we first opened it for update. We could ensure that the file has the new data recorded in it by doing a PUT using the fp of &. That would only write the buffer to the file. To update the FPDE's EOF value, either a CLOSE or a PUT fan,&& must be done. A CLOSE will also write out an updated buffer, if any.

Now let's go back to the second record and replace its single file item with several smaller ones. We'll do this using a couple of PUTS. Type in:

```
PUT 1,!19,,,"ITEM 1",3.14159*2;
PUT 1,*,,,"ITEM 3",4,10D2;
PUT 1,*,,,"LAST ITEM RECORD 2"; : GOTO 10
```

The system will respond with:

```
$ EOF TEST    198 % EOF RBA
152 ! NEXT RCD RBA    19 # REMRA
```

Once again, the next record pointer has the RBA of the record following the one we're processing, and the REMRA has the RBA of the record last processed. Note that all three PUT statements wrote items into the same record.

To show that the record has been updated type in:

```
GET 1,#,,A$,I,B$,J,K,C$; : PRINT A$,B$,C$,I,J,K : GOTO 10
```

The system will respond with:

```
ITEM 1    ITEM 3    LAST ITEM IN RECORD 2
6.28318 4 1000
0 $ EOF TEST    198 % EOF RBA
152 ! NEXT RCD RBA    19 # REMRA
```

That's pretty conclusive, isn't it? If we were to try to GET more data using the fp = *, we would find a "RECORD OVERFLOW" error staring back at us. We could, if we wanted to, add more data to this particular record, just as long as we didn't exceed its total original length of 131 bytes.

The only thing remaining to be done is to update the EOF value on disk. To do this, simply type in:

```
CLOSE
```

It should be noted, we could have used the statement:

```
PUT 1,&&
```

to update the EOF into the directory without closing the file. We could then have continued processing the file.

Once again, let's examine the file using SUPERZAP. Now you'll find SOR bytes at displacements 0, 13, 98 and AB. Examine The first record closely. The string marking byte (hex 8F) shows a length of 15 bytes. Adding hex F to the starting displacement of the string yields a result of hex 11. Looking at that displacement, you'll find the first of two bytes of hex 00. These are fill bytes which are skipped by the system as GETS are processed. If we were to try to retrieve two strings from the first record, as were there before our little updating session, we'd get a "RECORD OVERFLOW" error in response as there is now only one string item in the record. The system pads out a logical record with fill items when it finds that the data being written to the record has fewer bytes in it than were in the record to start with.

In the second record, starting at displacement hex 13, you'll find the SOR byte followed by a marking byte defining a string of 6 bytes. Counting down to the 7th byte from that marking byte, you'll find a marking byte defining a single precision numeric value. Five bytes further on you'll come across a marking byte defining another 6 byte long string. Seven bytes down from that byte is a marking byte defining an integer. The third byte beyond that is a double precision number marking byte. Nine bytes from there is the marking byte for the last item in the record, a hex 8A, defining a 10 byte long string. The remainder of the record following the string to displacement hex AB is filled with fill bytes. If it became necessary to replace record 2 with totally new data, the new record could take as many as 133 bytes, SOR byte included. All that is there right now would be replaced if the proper fp's were used.

The remainder of the record should be quite self-explanatory. The only differences between its first contents and now are the double precision number at displacement hex A2, and the new fourth record starting at AB and having its last byte at C5.

This discussion doesn't show all that can be done with MU files, of course. It is intended to show many of the abilities built into NEWDOS/80 BASIC file support. For those of you with data base experience, the partial-record I/O should look somewhat familiar. It is, after all, one of many data base abilities to update a single field in a record. Granted, NEWDOS/80 doesn't have the built-in file item security that data bases have; that is something you'll have to build into your systems as you see fit. But for now, you'll have to agree that NEWDOS/80's abilities are far superior to anything else available on the market.

For those of you getting into file processing for the first time, don't be daunted by the apparent complexities of the methods available to you. The best thing that you can do is to continue on with exercises similar to what we've just done here. As you practice, the concepts will seem to become easier to understand and work with.

MF FILES

Now that you've experimented with the MU file type and feel somewhat more comfortable with some of NEWDOS/80's capabilities, we'll go on now to experiment a little bit with the MF file type. Returning to chapter 8 you'll find that an MF file type is made up of marked items, and is record-segmented with all records having the same length. In other words, it is a marked item, fixed length record file. The length of the record is defined to the system by the lrecl operand of the OPEN statement.

Like the MU file type the MF file type can be updated with new data items on a record by record basis. The updating data need not be the same data type or length as the original data, nor does there have to be the same number of items in the updated record as there were to start with. You must be mindful of the file position being used during the updating of an MF file, just as you were with the MU file. The update can start in the middle of the record just as easily as at the beginning; the same fp controls are available to you for MF files as there were for MU files. Don't lose sight of the fact that when updating marked item files, all bytes from the current file position to the end of the record are re-written, whether you had really intended that to happen or not.

We'll use the same technique to experiment with the MF file as we used for the MU file. First we'll have to create a file for use as the experimental base. Enter the following BASIC program, and save it in case you need it again later.

```
10 OPEN "0", 1, "MF/DAT", MF, 20
20 PUT 1,,,"STRING1", "STR 2", "STR3";
30 PUT 1 "MAXIMUM STRING (19)";
40 I!=4 : I#=8 : I%=2
50 PUT 1,,I#,I!,I%;
60 PUT 1,,I#*10,I!*100,I%*1000;
70 CLOSE
```

Now run the program to create the file. When its done, run SUPERZAP using DFS to display sector 0 of the file just created. The first thing you'll notice is that there is no SOR byte at the beginning of the sector. That's because only MU files use them to define the start of records which are all presumed to have different lengths; other record-segmented file types have fixed length records so the system "knows" where each record begins. In the first byte is a marking byte describing a 7 byte long string. At displacement 8 is the marking byte describing a 5 byte long string, and at displacement E one describing a 4 byte string. Progressing down to displacement 13, where the next marking byte should be, you'll find a padding byte (00 hex). Remember that the records in the file we created are 20 (14 hex) bytes long. We wrote 3 items of 7, 5 and 4 bytes length respectively giving an aggregate byte count of 19; one fill byte is used to complete the 20 byte record.

The second record starts at displacement 14, where you'll find a marking byte describing a 19 (13 hex) byte long string. The one item is the entire record. The third record starts in displacement 28. You'll find marking bytes located at 28, 31 and 36 describing a double precision item, a single precision item and an integer respectively. This record has an aggregate data length of 17 bytes, and thus requires 3 padding bytes, which you'll find in displacements

39 through 3B inclusive. The fourth and last record we wrote has a data structure identical to that of the third record. Its marking bytes are located at displacements 3C, 45 and 4A; its padding bytes are in displacements 4D through 4F. The data beyond 4F is unpredictable. It is in fact whatever was in the sector before we created the file.

Return to BASIC and retrieve the location displaying program originally used when experimenting with MU files. It should read:

```
10 PRINT LOC(1)$; "$ EOF TEST "; LOC(1); "% EOF RBA"
20 PRINT LOC(1)!, "! NEXT RCD RBA ";
30 IF LOC(1)!=0 PRINT ELSE PRINT LOC(1)#; "# REMRA"
```

We'll use this program in the same way we did for the MU file experiments to show the results of GETS and PUTS on file position. The experiments we'll go through won't be as thorough as the ones done for the MU file. Instead they'll touch on the major differences between the two file types.

To start with, we'll open the file and examine the results of the LOC statements. Type in:

```
OPEN "I", 1, "MF/DAT", "MF", 20 : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST      80 % EOF RBA
0 ! NEXT RCD RBA
```

Except for the EOF RBA, the results are the same as for the MU file. The system is ready to process the record starting at displacement 0, the first logical record.

Now type this in:

```
GET 1,,,A$,B$; : PRINT A$, B$ : GOTO 10
```

The system will respond with:

```
STR 2      STR3
0 $ EOF TEST      80 % EOF RBA
20 ! NEXT RCD RBA    0 # REMRA
```

Notice that the last two items of the record were transferred. This is due to the null where the first variable name would normally reside (after the third comma).

From the current file position we can go back and transfer again the first two items of the record by using REMRA positioning. Type in:

```
GET 1,#,,A$,B$; : PRINT A$,B$ : GOTO 10
```

The system will respond with:

```
STRING1    STR 2
0 $ EOF TEST    80 % EOF RBA
20 ! NEXT RCD RBA    0 # REMRA
```

Nothing overly tricky there. As with MU files, we can continue processing the same record.

To do just that, type in:

```
GET 1,*,,C$; : PRINT C$ : GOTO 10
```

The system will respond with:

```
STR3
0 $ EOF TEST    80 % EOF RBA
20 ! NEXT RCD RBA    0 # REMRA
```

The fp "*" value tells the system to continue processing from where it left off on the preceding GET or PUT; in other words, from the current file position. If the GET had asked for two or more items, record overflow error would have occurred as the record, at that point, contained only one more item.

Now let's try processing the fourth logical record without first processing the second or third. Type in:

```
GET 1,!(4-1)*20,,J,K,L; : PRINT J; K; L : GOTO 10
```

The system will respond with:

```
80    400    2000
-1 $ EOF TEST    100 % EOF RBA
100 ! NEXT RCD RBA    80 # REMRA
```

Notice that the expression used in the !rba type fp specifies a value equal to 60. The numbers themselves represent the logical record number we really wanted, minus 1, times the record length. !rba positioning in a MF file, or a FF file too, is quite simple, as you can see. Just as something for you to do on your own, try the same statement as you just entered, using the rn form of fp instead of the !RBA form.

To do this, you should have changed the PUT statement to be:

```
GET 1,4,,J,k,l;
```

Now let's try some simple random updates to the records and check the results. Prepare the file for this by typing in:

```
CLOSE : OPEN "R", 1, "MF/DAT", "MF", 20 : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST    80 % EOF RBA
0 ! NEXT RCD RBA
```

Those are exactly the same results as when we opened the file for input. Again, no big surprise there.

As a starting point, let's replace the first record. Type in:

```
PUT 1,,,I$; : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST    80 % EOF RBA
20 ! NEXT RCD RBA  0 # REMRA
```

The responses show that the first logical record has been processed. You should be aware that even though the next record RBA shows a value of 20, the current file position is in fact equal to 1 as the above PUT replaced the entire contents of the record with a null string (an 80H marker byte only) and 19 bytes of zeroes, then repositioned the file back to the byte following the null string. If we were to write to the current file position using fp = *, the PUT's first marking byte would be placed in the second byte of the file.

Just for fun, let's add two fields to the record we just updated. Type in:

```
PUT 1,*,, "2",2; : GOTO 10
```

The system will respond:

```
0 $ EOF TEST    80 % EOF RBA
20 ! NEXT RCD RBA  0 # REMRA
```

We'll see the results of this last update in a moment.

Now, let's add two more records to the end of the file. Type in:

```
PUT 1,!%,, "RCD 5"; : PUT 1,,, "RECORD 6"; : GOTO 10
```

The system will respond with:

```
-1 $ EOF TEST    120 % EOF RBA
120 ! NEXT RCD RBA  100 # REMRA
```

The numbers indicate that the file is now six records long.

Close the file now, and enter the SUPERZAP program; use the DFS function to display sector zero of the file again. The records in the file begin at displacements 0, 14, 28, 3C, 50 and 64 respectively. The marking byte at displacement 0 describes a null string; the one at 1 a string 1 byte long and the one at 3 an integer. Notice that the remainder of the record has been padded with fill (00 hex) characters. The contents of the fifth and sixth

records should need no explanation. You should notice that the data beyond the sixth record was not modified by our little updating session. The system ignores this area of the sector as it is file space at and beyond the file's EOF and therefore not really part of the file.

As short as this session was in comparison to the one for MU files, you should now be aware that MF files are not at all hard to manage. Depending upon your own leanings, an individual record can be retrieved for update by either lrba positioning as shown in the example, or by using the record number itself On fp positioning).

MI FILES

Now we come to the last of the marked item files - the MI file type. Its most important differences from the MU and MF file types are:

1. MI files cannot be updated.
2. MI files have no system-recognizable record lengths.

These differences restrict this file type to being used for compact reference file only, as they can only be written to or extended, and later read again. Also, to get to any specific data group or item in a random-access fashion, !rba positioning (or its logical equivalents) must be employed.

Because you've seen marked item files in some detail by now, the experimental files accesses we've employed to this point will be quite limited and intended to amplify the differences in structure and access methods rather than similarities.

To start with, retrieve the program we used to create the MF file and change it to read as follows:

```
10 OPEN "O", 1, "MI/DAT", "MI"
20 PUT 1,,,"STRING1","STR 2","STR3";
30 PUT 1,,,"MAXIMUM STRING (19)";
40 I!=4 : I#=8 : I%=2
50 PUT 1,,I#,I!,I%;
60 PUT 1,,I#*10,I!*100,I%*1000;
70 CLOSE
```

Note that only line 10 of the program is changed from the MF file example.

Save the program if you wish, and run it. A user-segmented file will be created containing some 73 bytes of rather unlikely-looking data. Now exit BASIC and enter SUPERZAP, and use the DFS function to display sector zero of the file just created.

You'll see that there aren't any SOR marking bytes or padding items in the sector. There aren't any records in so far as BASIC is concerned, just a string of data items. The data in the file and its structure and organization are entirely the responsibility of the programmer. All you'll see in the sector is a series of contiguous marked data items. Good data design on the programmer's part demands that there be some rational, coherent data structure for the data items to be at all usable.

All there is in the file we created is unrelated data items. To access them sequentially would require the intimate knowledge we have: there are four strings and six numeric items. To access them randomly requires that we know the specific RBAs of the marking bytes. Otherwise at best, a "BAD FILE DATA" error will be raised by the system; at worst, it will return incoherent data.

Now, let's examine the SUPERZAP dump of the sector. The string marking bytes occur at displacements 0, 8, E and 13. The first set of numeric items have their marking bytes at 27, 30 and 35; the second set at 38, 41 and 46. We'll use these numbers (displacements, all in hex) in just a moment to access the data. By the way, the EOF RBA is 49.

Return to DOS BASIC at this time, and load the same location printing program as you used for MU and MF files. This program will aid in showing the lack of logical record support afforded to MI files by the system.

As usual, the file must be opened for access. Type in:

```
OPEN "I", 1, "MI/DAT", "MI" : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST    73 % EOF RBA
0 ! NEXT RCD RBA
```

As with other file types, the input mode open positions the system so that the next byte to be processed is the first byte in the file, if a (null) fp is used.

To show a different positioning resulting from open, and to extend the files besides, type in:

```
CLOSE : OPEN "E", 1, "MI/DAT", "MI" : GOTO 10
```

The system will respond with:

```
-1 $ EOF TEST    73 % EOF RBA
73 ! NEXT RCD RBA
BAD FILE MODE IN 30
```

This last message is due to the fact that the location printing program tries to print the REMRA value when it has just been marked invalid by the system as a result of the open itself. (The location printing program tries to display REMRA because the next record RBA is non-zero.)

The file is now in an output mode. To prove this we'll extend the file by three integer items. Type in:

```
PUT 1,,,-1,-2,-3; : GOTO 10
```

The system will respond with:

```
-1 $ EOF TEST    82 % EOF RBA
82 ! NEXT RCD RBA    73 # REMRA
```

Notice that the EOF RBA is 9 bytes higher in the file, and that the REMRA has the original EOF RBA value. In MI processing the REMRA is always set to the same value as the REMBA; they both equal the file position at the beginning of the GET or PUT data transfer.

Now, let's go back and reference a few of the data items. Type in:

```
CLOSE : OPEN "R", 1, "MI/DAT", "MI"  
GET 1,!19,,A$; : PRINT A$ : GOTO 10
```

The system will respond with:

```
MAXIMUM STRING (19)  
0 $ EOF TEST    82 % EOF RBA  
39 ! NEXT RCD RBA  19 # REMRA
```

The REMRA reflects the starting RBA of the GET, and the next record RBA points to the first of the numeric items. If no overriding fp were specified, that is where the next GET would start examining items for transfer.

To show this, type in:

```
GET 1,,,,J%,K#,I!; : PRINT J%; K#; I! : GOTO 10
```

The system will respond:

```
4  2  80  
0 $ EOF TEST    82 % EOF RBA  
65 ! NEXT RCD RBA  39 # REMRA
```

Notice that once again all the items in the IGEL are of a different numeric type than the file items being transferred to them. One of the marked item file's intrinsic powers is this numeric type conversion.

To show that in an MI file the REMRA and REMBA are the same, we'll have to do the same basic thing twice, with the appropriate fp characters. First type in:

```
GET 1,#,,I,J,K; : PRINT I; J; K : GOTO 10
```

Then enter:

```
GET 1,$,,I,J,K; : PRINT I; J; K : GOTO 10
```

In both cases, the system will respond with:

```
8  4  2  
0 $ EOF TEST    82 % EOF RBA  
56 ! NEXT RCD RBA  39 # REMRA
```

Q.E.D. Don't lose sight of the fact that this REMRA equals REMBA relationship is true at all times for field item, MI and FI files, and for MU, MF and FF files only when the GET/PUT data transfer starts at the beginning of a logical record.

Now, to show that an MI file can be extended after having been opened with mode "R", type in:

```
PUT 1,1%,15,-15; : GOTO 10
```

The system will respond with:

```
-1 $ EOF TEST 88    % EOF RBA  
88 ! NEXT RCD RBA   82 # REMRA
```

The EOF has been extended by 6 bytes as expected. The file is left positioned to continue adding data to the end of the file if fp = (null) or * are employed.

This about exhausts the experiments we can perform on MI files. On your own, you can try to update a single existing item. (You'll get a "BAD FILE MODE" error -- chapter 8 specifies that MI files cannot be updated.) If you are unsure of what will happen if an MI file has been opened with some mode, and a certain fp is specified in a GET/PUT, create the situation with a small file from BASIC's calculator mode and try it - it's the surest way to find out what does happen.

FF FILES

The fixed item file is different from the marked item file in several respects. To start with, it has no marking bytes for each item or record; all item description is taken from the IGEL, not the file. Because of this, if you describe a string item of 20 bytes to be read, that's exactly what will happen, even if the data written to the file originally was numeric. Also, it is required that numeric items written to file are read back as the same type; otherwise file synchronization is lost.

A second major difference is that fixed item files can be updated using true partial-record I/O. That is to say, a single field in a fixed item record may be updated without affecting any surrounding fields, whereas, in a marked item file, the field to be changed and all other fields to the end of the record had to be written.

A third significant difference is that the expressions in the IGEL cannot be anything more than variable names, with mandatory (len) prefixes for string items. This is due to the indeterminate type/length of an item resulting from an expression.

Fixed item files come in two types: FF files, in which all records have the same length, and FI files which have no BASIC detectable records. For the moment, we'll concern ourselves with only the FF type file.

As with the marked item discussions, we'll create an FF file, then experiment with it in "calculator mode". Enter the following program and save it if you wish. Then run it to create the FF file.

```
10 CLEAR 100
20 OPEN "O", 1, "FF/DAT", "FF", 20
30 PUT 1,%,40 : GOTO 50
40 (20)I$;
50 LSET I$="ABCDEFGHIJK"
60 PUT 1,,,(20)I$;
70 LSET I$="12345678901234567890"
80 PUT 1,,,(20)I$;
90 I%=2:I!=4:I#=8
100 PUT 1,,,(4)I$,I%,I!,I#;
110 I%=I%*10 : I!=I!*100 : I#=I#*1000
120 PUT 1,,,(4)I$,I%,I!,I#;
130 CLOSE
```

You will have noticed that this program is a little different than those used for the marked item files. For one thing, the string items are written from variables rather than literals in the IGEL proper. Additionally, no expressions as such were used to place numeric data on the file. In assigning values to the variable I\$, LSET was used instead of the (implied) LET. This latter was done to preserve the length of I\$ set up in the pseudo FIELD operation done in lines 30 and 40. This pseudo FIELD operation is not required if your program can live with the fact that variables providing string data to the file are NOT padded on the right while variables receiving data from the file are padded. Note too that all string items in the IGELS

have length prefixes. It's these prefixes that actually determine how many bytes of string data are to be transferred to/from the file, not the pseudo FIELD operation (refer to lines 100 and 120). Remember that the file string items are padded or truncated on the right as necessary to meet the length prefix's demands.

If we had elected NOT to use the pseudo FIELD function, the program could have been written:

```
10 CLEAR 1000
20 OPEN "O", 1, "FF/DAT", "FF",20
50 I$="ABCDEFGHIJK"
60 PUT 1,,,(20)I$;
70 I$="12345678901234567890"
80 PUT 1,,,(20)I$;
and so on
```

Now run SUPERZAP, and use the DFS function to examine the sector just written. You'll see that the first record (hex 11 bytes long) has the nine data bytes we had intended to transfer padded to 20 bytes with 14 blanks (the blank padding is due to the use of LSET in the first encoding above and due to the PUT in the second). The second record has no padding - the string item we wrote was twenty bytes long in the first place (had it been longer, the LSET in the first encoding would have truncated the variable I\$ on the right and the PUT in the second would have truncated the file item). The third and fourth records have identical formats: a four byte string, a two-byte integer value, a four-byte single precision value, an eight-byte double-precision value and two padding bytes. Again notice that there are no marking bytes to describe the type of the file item. The file's EOF is at displacement 80 (hex 50). Any bytes in the sector at or beyond this displacement were unmodified by the running of the program as those bytes are not part of the file.

Reload the program that was used to display the results of the LOC function as was used in the MU file experiments - we'll use it once again to demonstrate how file position is maintained.

To demonstrate the file's position after open, type in the following:

```
OPEN "I", 1, "FF/DAT", "FF", 20 : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST    80 % EOF RBA
0 ! NEXT RCD RBA
```

As expected, the system is positioned to process the next (first) record on the file.

To transfer the first record, type in:

```
GET 1,,,(20)A$; : PRINT LEN(A$); A$ : GOTO 10
```

The system will respond with:

```

20 ABCDEFIJK
0 $ EOF TEST    80 % EOF RBA
20 ! NEXT RCD RBA    0 # REMRA

```

As you can see, 20 bytes were transferred to the variable named in the IGET. We could just as easily have transferred a part of the record if we had wanted.

Just to show how this can be done, we'll assume that the record consists of 3 6-byte items and transfer them individually in separate GETS. Of course we'll have to use some special fp values to accomplish this task. Type in:

```

GET 1, #, (6)A$; : GET 1, *, (6)B$; : GET 1, *, (6)C$;
PRINT LEN(A$); A$ : PRINT LEN(B$); B$ : PRINT LEN(C$); C$ : GOTO 10

```

The system will respond:

```

6 ABCDEF
6 IJK
6
0 $ EOF TEST    80 % EOF RBA
20 ! NEXT RCD RBA    0 # REMRA

```

Here we've read 3 fields from the same record using as many GET statements to do it. This shows one of the freedoms of partial record I/O.

Another of the freedoms available to you is the ability to skip over bytes in a record to get to the ones you really want. We'll do that now with the second record. Type in:

```

GET 1, , (12)$, (4)A$; : PRINT LEN(A$); A$ : GOTO 10

```

The system will respond with:

```

4 3456
0 $ EOF TEST    80 % EOF RBA
40 ! NEXT RCD RBA    20 # REMRA

```

The 12 bytes we skipped could just as easily have been 6 integers as a 12 byte ASCII string. The point being made is that the system neither knows nor cares what data types or items are being skipped, only that Men) bytes are being skipped.

Now we'll make a slight error in processing the fourth record - we'll forget for a moment that was written with a 4-byte string at the start. Type in:

```

GET 1, 4, , I%, I!, I#; : PRINT I%; I!; I# : GOTO 10

```

The system will respond with:

```

12849 0 0
-1 $ EOF TEST    80 % EOF RBA
80 ! NEXT RCD RBA    60 # REMRA

```

Certainly not what we wrote! It does point out the need for consistent record description within FF (and FI, for that matter) files. Unlike a marked item file, in which this error would have been detected and reported, the fixed item processing demands that whatever is at the current file position be transferred to the named variable; no checks are done or can be done to prevent this type of error. (The reason for the zero values showing in the display for the single and double-precision numbers is that their exponent bytes were zero).

You'll notice that we're now also positioned at EOF, or at least apparently so. In fact the current file position, in so far as the system is concerned, is the 15th byte of the record. The LOC(fan)! returns the RBA of the start of the next sequential record to be processed; that is, the one which would be processed with an fp = (null).

Just to show that we are positioned at the 15th byte, type in:

```
FOR I=1 TO 6 : GET 1,*,,(1)A$; : PRINT ASC(A$); : NEXT
```

The system will respond with:

```
0 0 122 141 0 0
```

A little decimal-to-hex conversion will show the non-zero values to be the most significant mantissa byte and exponent byte respectively of the double precision number originally written as record 4.

Now let's go back and process record 4 correctly. Type in:

```
GET 1,4,,(4)A$,I%,I!,I#; : PRINT I$;I%,I!,I# : GOTO 10
```

The system will respond with:

```
1234 20 400 8000
-1 $ EOF TEST 80 % EOF RBA
80 ! NEXT RCD RBA 60 # REMRA
```

Just like it was written in the first place. You've noticed, of course, that the 4th record was processed using the rn form of the fp specification. When developing indices to fixed-length record files (FF, MF and field item) a couple of bytes can be saved by using integer items containing record numbers for the indices instead of single precision items containing the RBA value returned from some LOC function. Random access to a fixed-length record file is just as reliable using rn positioning as using RBA positioning, and perhaps a little easier to understand when examining an index item's contents.

Now let's close the file and open it for some examples of updating. Type in:

```
CLOSE : OPEN "R", 1, "FF/DAT", "FF", 20 : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST    80 % EOF RBA
0 ! NEXT RCD RBA
```

What we'll concentrate on is partial-record I/O. It's in this area that the fixed item files really have it over marked item files. Let's assume that the first and second records in our file have the same format: 4 5-byte long items each. Now let's update the 2nd item in the 1st record, and the last in the 2nd record. Type in:

```
A$="2ND" : PUT 1,1,,((2-1)*5)$,(5)A$; : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST    80 % EOF RBA
20 ! NEXT RCD RBA    0 # REMRA
```

Now, to update the 2nd record, type in:

```
A$="LAST" : PUT 1,2,,((4-1)*5)#,(5)A$; : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST    80 % EOF RBA
40 ! NEXT RCD RBA    20 # REMRA
```

You may have noticed that positioning to the field in the record was done by computing the number of bytes to be skipped. In the 2nd record, the positioning to the last 5-byte field didn't simply skip over the preceding 15 bytes, but nulled them out in the process. We'll see the effects of this later.

Now let's update the integer items in the 3rd and 4th records. Type in:

```
I%=-50 : PUT 1,3,,(4)$,I%; : PUT 1,4,,(4)$,I%;
GET 1,3,,(4)J$,J%,J!,J#; : PRINT J$,J%;J!;J# : GOTO 10
```

The system will respond with:

```
1234      -50  4  8
0 $ EOF TEST    80 % EOF RBA
60 ! NEXT RCD RBA    40 # REMRA
```

The first line of the response shows that our update affected only the one field we wanted to mess with - the other fields in the record were not modified. In MF and MU files having similar records (allowing for marking bytes), the integer, single-precision and double-precision values would all have had to be specified in the IGEL in order to have updated just the integer. That statement isn't quite complete: the single and double-precision numbers would have to have been read first to maintain the correct values; also, they could have been written back individually using the various fp values. Again, here in an FF file, we had only to skip over the bytes we wanted to, and write the single field to be modified.

While in the "R" mode, any NEWDOS/80 file may be extended. To show this feature, type in:

```
PUT 1,6,,J%,J%,J%; : GOTO 10
```

The system will respond with:

```
-1 $ EOF TEST    100 % EOF RBA  
100 ! NEXT RCD RBA    80 # REMRA
```

In this example we entirely skipped over the 5th record of the file. The system, in order to maintain the necessary record orientation, created and wrote a 5th record containing only nulls before writing the 6th record as we specified.

Now close the file, run SUPERZAP, and use the DFS function as before to examine the 1st sector of the file. You'll notice that the 2 string records (numbers 1 and 2) have been updated as required, that the integer items in the 3rd and 4th records both read CEFF, that the 5th record (displacements 50-63 hex inclusive) is all nulls, and that the 6th record contains 3 repetitions of CEFF hex, followed by 14 nulls.

FI FILES

We now come to the last of NEWDOS/80's unique file types: the FI file. Like the MI type file, it is a user segmented file; and like the FF type file, it is made of fixed items, rather than marked items. Unlike the MI type file, the FI file can be updated. This attribute makes it a little more powerful in its application than the MI type file.

As we have done with each file type up to this point, we'll create a file by running a BASIC program, then experiment with that file from the BASIC calculator mode. To create the file, enter, save and run the following program.

```
10 OPEN "O", 1, "FI/DAT", "FI"
20 A$="1ST STRING" : B$="STR 2"
30 I%=2 : I!=4 : I#=8
40 PUT 1,,,(15)A$,(6)B$,I%,I!,I#;
50 I%=I%*-1000 : I!=I!*-100 : I#=I#*-10
60 PUT 1,,,(15)B$,(6)A$,I%,I!,I#;
70 CLOSE
```

The first thing you'll have noticed is that we never did a pseudo FIELD operation Up = X) as we did for the FF type file. This is because it isn't absolutely necessary. BASIC will allocate the strings on GETS as it needs to, and the file support will pad/truncate the string file items as needed to make them fit the length specified by the IGEL item prefix.

The second thing to notice is that the PUTS both put out data groups having identical formats: a 15-byte string, a 6-byte string, an integer, a single precision value and a double precision value. In a larger file, such a consistent data group format would make it eligible for an FF type file, as all data groups would have the same length and structure.

Load the program used to print the LOC function results used in all previous experiments, and type in:

```
OPEN "R", 1, "FI/DAT", "FI" : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST    70 % EOF RBA
0 ! NEXT RCD RBA
```

As expected, the file is positioned so that the first GET or PUT will begin processing at the first byte of the file if fp = (null) or * is specified. This would be the case for all open modes except "E", which would position the file to the EOF RBA.

Knowing the data structure of the two groups that we wrote makes it reasonably easy to access the second group via RBA positioning. Type in:

```
GET 1,135,,(15)A1$,(6)A2$,I%,I!,I#;
PRINT A1$, A2$, I%; I!; I# : GOTO 10
```

The system will respond with:

```
STR          1ST ST      -2000 -400 -80
-1 $ EOF TEST   70 % EOF RBA
70 ! NEXT RCD RBA   35 # REMRA
```

By processing all the data in the second data item group, the file is positioned at EOF as the LOC(fan)\$ shows.

FI files can be extended in the same manner as FI files. Let's do just that, and leave an area of 10 bytes between the current file position and the new data. Type in:

```
L=LOC(1)! : J=EXP(1) : PUT 1,,, (10)#,J; : GOTO 10
```

The system will respond with:

```
-1 $ EOF TEST   84 % EOF RBA
84 ! NEXT RCD RBA   70 # REMRA
```

The variable L contains the file location of the 10 padding bytes we wrote. We'll use that in a minute. Notice that in the PUT statement, the variable J was not suffixed by a type character. J has the default type of single precision floating point and 4 bytes were written into the file. Though using explicit type suffix characters in IGELs is not required as it was for NEWDOS/80 version 1, it is highly recommended that you do so.

Now, let's go back and put something in that padding area we just wrote. We'll use RBA positioning again to get to that area of the file. Type in:

```
A$="ABCDEFGFG" : PUT 1, ! L, , (4) A$, L! ; : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST   84 % EOF RBA
78 ! NEXT RCD RBA   70 # REMRA
```

Pay special attention to the spacing on that last PUT statement. It shows the freedom you're allowed when entering a program. Depending on your own leanings, the spacing may or may not make the program more readable. Feel free to use spacing or not as you see fit.

Now let's go back and read some of the data we've written to the file. Type in:

```
GET 1,158,,I!,I#,(4)B$,K!;
PRINT I!; I#; B$; K! : GOTO 10
```

The system will respond with:

```
-400 -80 ABCD 70
0 $ EOF TEST    84 % EOF RBA
78 ! NEXT RCD RBA    58 # REMRA
```

We read the data with the proper data types for what was located at the starting file position, so the results of the PRINT are normal. Needless to say, if we were off by even 1 byte in our positioning the results would have been rather different.

To show the disaster which could befall the unwary programmer, lets malposition the file and repeat the last transfer. Type in:

```
GET 1,157,,I!,I#,(4)B$,K!;
PRINT I!; I#; B$; K! : GOTO 10
```

The system will respond with:

```
2.36125E+21 2147483648.000008 xABC 6.01858E-36
0 $ EOF TEST    88 % EOF RBA
77 ! NEXT RCD RBA    57 # REMRA
```

Nasty, isn't it? But the system did just what we told it to do - because it's an FI type file, it couldn't protect us from ourselves.

If you've gotten this far, you should have a fair idea of how to manipulate marked item and fixed item files. As the support for TRSDOS BASIC's field item files and print/input files has not changed in NEWDOS/80, excepting for allowing field item files to have a standard record length of other than 256, no experiments were provided here to familiarize you with them. If you haven't done so lately, go back and read chapter 8. You'll see that indeed the support for the old TRSDOS file types has been extended. With the experience you've received here, you'll be able to generate your own experiments for those extensions. Good luck to you!

APPENDIX B

The purpose of this appendix is to give some examples of marked item and fixed item file usage and to give different explanations than were offered in chapter 8 and Appendix A. Chapter 8 contains the specifications for the I/O enhancements to BASIC. This appendix hopes to give enlightenment but is not the specifications; chapter 8 is!!!

Throughout this appendix (as well as the whole manual) we shall refer to the file types by their short names. The reader should refer now to the glossary in chapter 10 for the definitions of MI file, MU file, MF file, FI file and FF file. This appendix will also refer to other terms such as IGEL, RBA, REMRA, REMBA, etc. which are defined in the glossary or in chapter 8.

Most of the examples given in this appendix deal with MU files and FF files since these two types will be the most commonly used by the programmers.

We have tried to make the examples as much alike as possible or practical to make it easier for the reader to spot the differences.

Since we are basically interested in demonstrating the use of the files (an exception is the demonstration of the uses of CMD"O", BASIC's in-memory sort), we do not provide the routines which actually use or generate the data to be read from or sent to the files. The programmer is assumed to provide these routines if he/she wishes to use these examples in live situations.

In all these examples, each named variable corresponding to a file item is suffixed with an explicit type symbol (\$, %, 1 or #) (See line 120 of Example 3). This is done so that the reader will know exactly what type of data is being read from or written to the file. We STRONGLY recommend that in your own IGELS that you do the same; otherwise it is quite possible that you can severely damage a file by the implied type not being as you thought you remembered it. Use of explicit type symbols was required in version 1 for IGELS used in fixed item file processing, but that is not so in version 2. An example of an IGEL that does not use explicit type symbols is to change two lines in Example 7 to be:

```
10 CLEAR 2000: DEFSTR N,S: DEFINT A,I: DEFSNG F: DEFDBL D
120 (20)NM,AN,AM!,DT,(15)ST,IG,FP,DP;
```

Remember, we STRONGLY recommend the suffixing of type symbols to the variable names in IGELS.

The operation of a GET or a PUT proceeds in two phases:

1. The file positioning phase. In this phase, the file is positioned according to the second parameter, the file positioning parameter, of the GET or PUT statement. At the end of this phase, for certain types of positioning parameters, the file location values REMRA and REMBA are saved for possible future use when the subsequent positioning parameter for that filearea is # or \$ (see section 8.10).
2. The data transfer phase. In this phase, data is transferred between the file and the variables named in the IGEL.

Example 1. Write records sequentially to a MU file.

MU files are intended as an alternative to print/input files. AMU file tends to use less disk space than a print/input file, can be updated with some restrictions, can be indexed into via the !rba positioning parameter, and does not need the ;", " ; character sequence to separate strings during writes to the file.

```
10 CLEAR 2000
20 OPEN "O",1,"XXX/DAT:1","MU"
30 GOSUB 10000 'build data for record
40 IF RN% = 0 THEN CLOSE: END 'end of run
50 PUT 1,,NM$,AN%,AM!,DT#,ST$,IG%,FP!,DP#;
60 GOTO 30
```

The file is opened for sequential output of records whose individual lengths vary depending upon the size of the two strings contained in each record.

The file positioning parameter in the PUT statement is null, indicating that each execution of that PUT writes the next sequential record.

The programmer supplies the routine at 10000 to generate the data for the records. If no more records are to be created, set RN% = 0. Otherwise set RN% not 0 and put into the 8 variables NM\$, AN%, AM!, DT#, ST\$, IG%, FP! and DP# the data that is to be transmitted to the file.

The IGEL in this example is contained within the PUT statement and consists of 8 expressions (in this case, all named variables). The variable or expression associated with each record item is separated from its neighbor by a comma. The IGEL is terminated by a semicolon.

The full contents of each of the strings NM\$ and ST\$ is sent to the file, with each preceded by one or two string marker bytes. The second marker byte is used for strings 128 to 255 characters in length.

Each of the integers AN% and IG% is represented in the file as 3 bytes, a 72H marker byte followed by the 2 bytes of the binary integer value in the same format as used by BASIC.

Each of the single precision floating point numbers AM! and FP! is represented in the file as 5 bytes, a 73H marker byte followed by the 4 byte binary single precision floating point value in the same format as used by BASIC.

Each of the double precision floating point numbers DT# and DP# is represented in the file as 9 bytes, a 74H marker byte followed by the 8 byte binary double precision floating point value in the same format as used by BASIC.

Using the IGEL in the PUT statement to compute the minimum and maximum record lengths, the minimum length of a record in this file will be 37 bytes (both strings are null and including the SOR byte) and the maximum length of a record will be 549 bytes (both strings have 255 characters).

Example 2. Read records sequentially from a MU file.

```
10 CLEAR 2000
20 OPEN "I",1,"XXX/DAT:1","MU"
30 IF EOF (1) THEN END
40 GET 1,,,NM$,AN%,AM!,DT#,ST$,IG%,FP!,DP#;
50 GOSUB 10000      'process the record's data
60 GOTO 30
```

This example is the opposite of Example 1, using the same IGEL and named variables. The data records of the file are successively read and processed. The programmer supplies the routine at 10000 to do what he/she wishes with the data.

The file positioning parameter in the GET statement is null, meaning that each execution of that GET reads the next sequential record in the file.'

The EOF(1) function returns a true condition when the file position of the next record is exactly at file EOF.

Example 3. Sequentially read and update the records of a MU file.

```
10 CLEAR 2000
20 OPEN "R",1,"XXX/DAT:1","MU"
30 IF EOF(1) THEN END
40 GET 1,,120      'read the next sequential record
50 GOSUB 10000      'update the record's data
60 PUT 1,#,120      'rewrite the record back to the file
70 GOTO 30
120 NM$,AN%,AM!,DT#,ST$,IG%,FP!,DP#;
```

The same file created in Example 1 is used in this example. The file is opened for both input and output operations. Records are read sequentially from the file into the BASIC variables, zero or more of those variables are updated by the programmer supplied routine at statement 10000. Upon return from that routine, the record is written back to the file.

The file positioning parameter in the PUT statement is the character #. For this example, at the start of each execution of that PUT statement, the file is repositioned back to the start of the record read by the GET, causing the PUT to update that record.

Both the GET and the PUT statement use the same IGEL which is located at text line 120. This IGEL is identical to that used in examples 1 and 2 except that instead of being contained within the GET or PUT statement, it is contained in a separate text line with the GET and PUT statements containing that line number as their third parameter.

An error will be declared if the PUT statement finds the new length of a record exceeds the length originally assigned to that record during Example 1. This will only occur when the sum total of the file space used by the record's string items exceeds that of what the strings originally occupied plus any null space included in the original record (insertable using the (lend function, see section 8.4.3.4). The numeric values may be updated

without concern as they always occupy the same amount of file space. Thus, if a string item is to be updated in a MU file, the string's resulting length should not be increased; it can be, but be careful.

Example 4. Read in, sort in memory and write back out a MU file.

```
10 CLEAR 10000: DEFINIT I
15 DIM NM$(200),AN%(200),AM!(200),DT#(200)
17 DIM ST$(200),IG%(200),FP!(200),DP#(200),IX%(200)
18 DIM IX%(200)
20 IX=0: OPEN "I",1,"XXX/DAT:1","MU"
30 IF EOF(1) THEN 80
40 IX=IX+1: IF IX > 200 THEN PRINT "TOO MANY RECORDS": END
50 GET 1,,60: GOTO 30
60 NM$(IX),AN%(IX),AM!(IX),DT#(IX),      'IGEL 1st line
70 ST$(IX),IG%(IX),FP!(IX),DP#(IX);      'IGEL last line
80 IF IX = 0 THEN PRINT "EMPTY FILE": END
90 CMD"O",IX,*IX%(1),AM!(1),NM$(1)
100 CLOSE: OPEN "O",1,"XXX/DAT:1","MU"
110 IY = IX: FOR IZ = 1 TO IY
120 IX = IX%(IZ): PUT 1,,60
130 NEXT IZ: CLOSE: END
```

The MU file XXX/DAT:1 of 1 to 200 records is read into 8 arrays.

The records are then indirectly sorted at line 90 using BASIC's array sort using the IX% array as the integer indirect array. The sorting criteria is ascending order, first by the AM! values and then by the NM\$ values. During the sort, the integer IX% array is set up to contain sequentially in the sorted order the index values into the other arrays.

It should be noted that the sort changed nothing in the record arrays AM! and NM\$. The IX% array was initialized to point each successive element to successive elements of the AM! array. As the sort proceeded, the elements in the IX% array were moved around to conform to the sort order.

It should further be noted that though the file records span across 8 arrays, the sort saw only the two of them (AM! and NM\$) that provided the sort data.

After the sort the records of the file are written out in sorted order. Since the same file was used to store the sorted records, the user must be sure to preserve a backup copy of the original file in case an error occurs during the output of the sorted records.

This example demonstrates that IGELs can contain array named variables and that an IGEL may span multiple text lines (lines 60 and 70).

Example 5. Write records sequentially to a FF file.

FF files are intended as an alternative to field item files (TRSDOS random files). The FIELD statement is not used with FF files; though the user may wish to use the pseudo FIELD function specified in section 8.11. LSET and RSET are not used in FF file processing to set up the variables making up the record, though if the user has set the strings to the specified lengths via the pseudo FIELD function, he/she may wish to use LSET or RSET to maintain a string variable at that length. LSET and RSET must never be used for numeric variables. For FF files, MKD\$, MKI\$, MKS\$, CVD, CVI and CVS are not used.

Each string variable in the IGEL must be prefixed with the length the string item will have in the file, and regardless of the number of characters in the variable's string at the time of the PUT, the corresponding string in the file will be truncated on the right or padded on the right with spaces to make up the required number of characters. During the PUT, the string variable is NOT changed; only the file item is.

```
10 CLEAR 2000
20 OPEN "O",1,"XXX/DAT:1","FF",63
30 GOSUB 10000      'build data for record
40 IF RN% = 0 THEN CLOSE: END      'all done
50 PUT 1,,,(20)NM$,AN%,AM!,DT#,(15)ST$,IG%,FP!,DP#;
60 GOTO 30
```

The file is opened for sequential output of records each 63 bytes long.

The file positioning parameter in the PUT statement is null, indicating that each execution of that PUT writes the next sequential record.

The programmer supplies the routine at 10000 to generate the data for the records. If no more records are to be generated, set RN% = 0. Otherwise set RN% non-zero and load the 8 variables NM\$, AN%, AM!, DT#, ST\$, IG%, FP! and DP# with the data that is to be transmitted to the file.

The IGEL is contained within the PUT statement and consists of 8 named variables. The name variable associated with each record item separated from its neighbor by a comma. The IGEL is terminated by a semicolon.

Each of the strings NM\$ and ST\$ is represented in the file by the number of characters specified by the variable's prefix in the IGEL. For each PUT executed by this example, the current contents of NM\$ are sent to the file. If the NM\$ string has more than 20 characters, the excess characters on the right are dropped from the file item, not from NM\$. If the NM\$ string has less than 20 characters, the file item, not NM\$, is padded on the right with spaces to make the file item 20 characters long. The same concept holds in restricting to 15 characters the file item associated with the ST\$ string.

Each of the integers AN% and IG% is represented in the file by 2 bytes in the same format as used by BASIC.

Each of the single precision floating point numbers AM! and DF! is represented in the file by 4 bytes in the same format as used by BASIC.

Each of the double precision floating point numbers DT# and DP# is represented in the file by 8 bytes in the same format as used by BASIC.

Example 6. Read records sequentially from a FF file.

```
10 CLEAR 2000
20 OPEN "I",1,"XXX/DAT:1","FF",63
30 IF EOF(1) THEN END
40 GET 1,,(20)NM$,AN%,AM!,DT#,(15)ST$,IG%,FP!,DP#;
50 GOSUB 10000      'process the record's data
60 GOTO 30
```

This example is the opposite of Example 5, using the same IGEL and named variables. The data records of the file are successively read and processed. The programmer supplies the routine at 10000 to process the data.

The file positioning parameter in the GET statement is null, meaning that each execution of that GET reads the next sequential record in the file.

After each record read, AM\$ contains a 20 character string and ST\$ contains a 15 character string.

Example 7. Sequentially read and update the records of a FF file.

```
10 CLEAR 2000
20 OPEN "R",1,"XXX/DAT:1","FF",63
30 IF EOF(1) THEN END
40 GET 1,,120      'read the next sequential record
50 GOSUB 10000      'update the record's data
60 PUT 1,#,120      'rewrite the record back to the file
70 GOTO 30
120 (20)NM$,AN%,AM!,DT#,(15)ST$,IG%,FP!,DP#;
```

The same file created in Example 5 is used here. The file is opened for both input and output operations. Records are read sequentially from the file into the BASIC variables, zero or more of those variables are updated by the programmer supplied routine at statement 10000. Upon return from that routine, the record is written back to the file.

The file positioning parameter for the PUT statement is the character A At the start of each execution of the PUT in this example, the file was repositioned back to the start of the file record read by the GET (in more complicated words, to the REMRA), thus causing the PUT to write that record.

Both the GET and the PUT use the IGEL starting at line 120.

After each GET, NM\$ contains a 20 character string and ST\$ contains a 15 character string. During the programmer supplied processing, the lengths of one or both of the strings may change. During the PUT, the file item corresponding to AM\$ is set again to a length of 20 characters, with space padding or truncation taking place on the right as necessary. The same

concept applies to the 15 character file item corresponding to ST\$. Remember, AM\$ and ST\$ are not changed by the PUT.

Example 8. Randomly read and optionally update the records of a FF file.

```
10 CLEAR 2000
20 OPEN "D",1,"XXX/DAT:1","FF",63
30 GOSUB 10000      'determine which record to read
40 IF RN% = 0 THEN END 'end if no more
50 GET 1,RN%,120    'read that record
60 GOSUB 15000      'optionally update the record's data
70 IF RN% <> 0 THEN PUT 1,RN%,120 'if required, write the record
80 GOTO 30
120 (20)NM$,AN$,AM!,DT#,(15)ST$,IG%,FP!,DP#;
```

This example is similar to Example 7 excepting that the record reads are done randomly and the programmer can elect not to update the record.

The file created in Example 5 is used here. For each record, the processing is as follows:

The programmer supplied routine at line 10000 determines which file record is to be looked at next. On return, RN% contains the desired record number; if RN% = 0, the run is ended.

The record is read, using RN% as the file positioning parameter to specify which record is wanted.

The programmer supplied routine at line 15000 looks at the record's data and optionally changes 1 or more variables associated with the record. On exit from the routine, RN% is set to zero if the record is not to be updated; otherwise RN% is unchanged.

If RN% is not zero, the record is rewritten to the file using RN% as the file positioning parameter. Note, the file positioning parameter for the PUT could have been the character

Note the use of "D" rather than "R" as the 1st parameter in the OPEN statement. "R" could have been used, but "D" prevents the file from being extended if for some reason RN% was changed before the PUT statement to be a record number beyond the range of the file.

Example 9. Sequentially write records to a MU file and sequentially write records to a FF file that serve as an index into the MU file.

There are many cases where the user has a huge file with each record having strings of varying lengths, does not want the string padding or truncation that is done by field item or fixed item files and yet still wants to be able to randomly access the file, and to a limited extent be able to update that file. Using a MU file as the main file and an FF file as an index file, the user can achieve these objectives.

```

10 CLEAR 2000
20 DIM AN%(4000),RB!(4000)      'two arrays to hold index data
30 OPEN "O",1,"XXX/DATA","MU"  'open the main data file
40 RC% = 0
50 GOSUB 10000                  'create next record's data
60 IF RN% = 0 THEN 105          'done with main file
70 RC%=RC%+1: IF RC% > 4000 THEN PRINT "FILE TOO LARGE": GOTO 105
80 RB!(RC%) = LOC(1)!          'save RBA of next record
90 PUT 1,,NM$,AN%(RC%),AM!,DT#,ST$,IG%,FP!,DP#;
100 GOTO 50
105 CLOSE
110 IF RC% = 0 THEN PRINT "NO DATA RECORDS": END
120 CMD"O",RC%,AN%(1),RB!(1)    'sort index data
130 OPEN "O",1,"XXX/NDX:1","FF",6 'open index file
140 FOR X = 1 TO RC%
150 PUT 1,,AN%(X),RB!(X);      'write index record
160 NEXT X: CLOSE: END

```

This example could have been programmed to write alternately one record to each of the two files. However, since both are on the same drive, the drive arm would be constantly in motion and execution would take, or at least seem to take, forever. Therefore, the index file's data is stored into arrays to be written out after the main file has been completely written.

For this example the AN% array is assumed to hold account numbers and for each main data file record, the account number is unique.

The program proceeds as follows:

For each main data file record:

The programmer supplied routine at 10000 set RN% = 0 if no more main data file records are to be created. Otherwise it sets RN% non-zero and creates the record's data by storing the values in the proper variables, including the account number into its array.

The RBA of where that record is to be placed in the main data file is determined by the line 80 LOC(1)! function and is stored into the RBA array RBI.

The record is written to the main data file.

The two arrays AN% and RBI are directly sorted. Since this is a direct sort, both arrays are physically arranged in the sort order, which is in ascending order of account number. Note, though ascending order of RBA is the secondary sort criteria, since the account numbers are unique, the RBA values are never checked.

The index file is created by writing the index records sequentially from the arrays, which are in ascending order of account number.

Exactly the same results would have been attained had text lines 80 and 90 above been written as:

```
80 PUT 1,,,NM$,AN%(RC%),AM!,DT#,ST$,IG%,FP!,DP#;
90 RB! = LOC(1)#          'RBA where the record was placed
```

Example 10. Randomly read and optionally update the records of an indexed MU file.

```
10 CLEAR 2000
20 DIM AN%(4000),RB!(4000)
30 RC% = 0: OPEN "I",1,"XXX/NDX:1","FF",6          'open index file
50 IF EOF(1) THEN 100
60 RC% = RC% + 1
70 IF RC% > 4000 THEN PRINT "INDEX TOO LARGE": END
80 GET 1,,,AN%(RC%),RB!(RC%);          'read index data into arrays
90 GOTO 50
100 CLOSE: IF RC% = 0 THEN PRINT "NO ACCOUNTS": END
110 OPEN "D",1,"XXX/DATA","MU"          'open main data file
120 GOSUB 10000      'determine which account record wanted
130 IF RN% = 0 THEN END          'if req, end the run
140 FOR X = 1 TO RC%
150 IF RN% = AN%(X) THEN 170      '.search index for account #r match
160 NEXT X: PRINT "BAD ACCOUNT NUMBER": GOTO 120
170 GET 1,!RB!(X),300          'read the account record
180 IF AN% <> RN% THEN PRINT "BAD DATA FILE": END
185 GOSUB 15000      'display data, receive back updates
190 IF RN% <> 0 THEN PUT 1,#,300      'if required, re-write record
200 GOTO 120
300 NM$,AN%,AM!,DT#,ST$,IG%,FP!,DP#;
```

It is assumed that you, the programmer, have an application that is basically data retrieval for display to the terminal operator and in some cases, update information is received from the terminal operator to alter information already in the main data file.

The two files created in Example 9 are used in this example. The index file is opened first and its contents read into the two arrays AN% and RB!

For each record, the processing is as follows:

The programmer supplied routine at 10000 queries the terminal operator to determine the account number. On return from that routine RN% contains the account number; if zero, the run is to end.

The account number array AN% is searched for the matching account number. If not found, an error message is displayed.

The account record is read from the main data file, using the account record's RBA value from the RB! array as the file positioning

parameter. RN% is then compared against the AN% value from the file and if not equal, an error message is displayed and the run terminated.

The programmer supplied routine at 15000 displays the account data to the terminal operator, and, if required, accepts back the new data for the fields being updated. If the record is not to be updated, set RN% = 0.

On return, the record is re-written to the file if RN% is non-zero. The PUT uses file positioning parameter # which repositions the file to the start of the record (again, in other words, the # causes file positioning to REMRA).

Example 11. Sequentially write different type records to a MU file.

A programmer's application may use different types of records. Normally, the individual record types would be stored in different files, but if a record of one type corresponds to one or more records of one or more other record types the programmer may want to store all of these associated records together in the file in order to minimize the time needed to access them. MU, MI and FI files readily allow this mixing of record types whereas field item, MF and FF files do so only if the records are all the same length.

An example of this mixture of record types might be an insurance account which can have the account main record, one or more records of the individuals covered by the account, one or more records of payments and one or more records of claims, each of the 4 record types having different formats.

In the example below, three different record types are used; the main record and two subsidiary types. There is no correlation between these three record types and anything in the real world; all we are trying to do is demonstrate how a MU file can contain the multiple type records.

In the example below, each set of records consists of one type 1 record and a mixture of zero or more type 2 and 3 records.

```
10 CLEAR 2000
20 OPEN "O",1,"XXX/DAT:1","MU"
30 GOSUB 10000      'create type 1 record's data
40 IF RN% = 0 THEN CLOSE: END      'all done
50 PUT 1,,,"1",NM$,AN$,AM!,DT#,ST$,IG%,FP!,DP#;  'write type 1 record
60 GOSUB 11000      'create type 2 or 3 record's data
65 IF X$ <> "2" THEN 90
70 PUT 1,,,"2",SA$,SB$,LN$,PD!;      'if type 2 record, write it
80 GOTO 60
90 IF X$ <> "3" THEN 30
100 PUT 1,,,"3",SJ$,DF#,IP%,IA$,FG!;      'if type 3 record, write it
110 GOTO 60
```

The programmer supplied routine at 10000 sets RN% = 0 if no more records are to be created. Otherwise it sets RN% non-zero and creates the type 1 record's data. For each set of records, there is one and only one type 1 record.

The programmer supplied routine at 11000 creates the type 2 or type 3 record's data, whichever comes next. On exit from that routine X\$ contains the record type flag or if neither "2" or "3", it indicates the end of the series of records. The type 2 and type 3 records are intermixed on the file following the associated type 1 record. For a given type 1 record, there need not be any type 2 or 3 records.

Each PUT statement contains its own IGEL. Note, in each of these IGELs the first entry is an expression rather than a named variable. It could have been a named variable containing the record type character, but expressions were used instead to demonstrate that the IGELs used for writing to marked item files can contain expressions as well as named variables, hence the reason why its called an IGEL (item group expression list) instead of an IGVL (item group variable list).

Example 12. Sequentially read and optionally update records from a MU file containing multiple record types.

The file created in Example 11 is used here. In this example we will demonstrate the partial record read feature of marked item files (also available for fixed item files). The first three, the 5th and 7th items of the type 1 record will be read, type 2 records will be skipped, and type 3 records will be processed entirely, including optionally being updated.

```

10 CLEAR 2000
20 OPEN "D",1,"XXX/DATA","MU"
30 IF EOF(1) THEN END
40 GET 1,,,RT$;      'read record type character
50 IF RT$ <> "1" THEN PRINT "BAD RECORD TYPE": END
60 GET 1,*,,NM$,AN%,DT#,,IG%;      'read selected type 1 record items
70 IF EOF(1) THEN END
80 GET 1,,,RT$;      'read next record type character
90 IF RT$="2" THEN 70 'bypass type 2 records
100 IF RT$ <> "3" THEN 50
110 GET 1,*,200      'read in rest of type 3 record
120 GOSUB 11000      'process type 3 record's data
130 IF RN% <> 0 THEN PUT 1,$,200 'if required, re-write type 3 record
140 GOTO 70
200 SJ$,DF#,IP%,IA%,FG!;
```

The GET statements at lines 40 and 80 read only one item of the next record, and the file is left positioned at the 2nd item of the record. Both REMRA and REMBA point to the record's 1st item.

The GET statements at lines 60 and 110 start where the line 40 or 80 GET left off. The line 60 and 110 GETS do NOT advance to the next record before inputting data. However, before inputting the data REMRA is set to point to the positioning point which is the 2nd item of the record. Thus, if the line 130. PUT is executed, the PUT's reposition parameter will reposition the file to REMBA which is pointing at the record's 2nd item. REMRA is not changed by the line 60 or line 110 GET statements.

The line 60 GET reads the 2nd, 3rd, 5th and 7th items of the type 1 record and leaves the file positioned at the 8th item. The 4th and 6th items were skipped, and the next execution of the line 80 GET will skip the remainder of the type 1 record items (the 8th and 9th).

The remainder of the item in the type 2 records are skipped over by the next execution of the line 80 GET.

The line 110 GET reads the rest of the type 3 record, the 2nd through 6th items.

The programmer supplied routine at line 11000 does whatever processing is needed for the type 3 record, using its data plus that extracted from the type 1 record. If the type 3 record is to be updated, RN% is set non-zero; otherwise it is set equal to 0.

On return for that routine, if RN% <> 0 the type 3 record is re-written to the file. Note that only the 2nd through the last items were written back to the file. The first item was not changed as the file positioning done for the line 130 PUT was to the REMBA position which was the file position existing during the line 110 GET immediately after its positioning was done and before any items were inputted.

Remember that in updating a MF and MU record, once an item is written back to that record, all items following it in the record must also be written back if those items are to remain part of the record. It is not necessary they all be written by the same PUT statement, but they must all be written; for each PUT that updates only part of a MU file record fills with null bytes all of the record's bytes, if any, following the last item written.

For record segmented files, EOF compares the location of next record against the EOF. For the type 3 record processing above, the position where the line 130 PUT leaves off and the position of the next record are the same. For the type 1 and 2 records, the GET statements left the file positioned to the 8th and 2nd items respectively. In these cases, EOF computes the position of the next record and uses that value in its compare against EOF.

Example 13. Sequentially write records to a MF file (marked item file of fixed length records).

A programmer's application may require full update capability for a file that contains strings. Since a MU file cannot guarantee record update success when strings are being lengthened, the programmer must go to either field item file, a FF file or lastly, a MF file. The relative merits of the field item and the FF file have already been discussed, so we will concern ourselves here only with the relative merits of the FF and MF files.

The advantage of MF files is that string items are not padded with blanks to fill out the item to the maximum length allowed it. Each string item is written with the number of characters it needs, up to but not exceeding the maximum length allowed it. Then, at the end of the record, if unused record space exists, the record is filled out with null bytes which during the read of a MF file, the program never sees. Though most of the time padding bytes do not bother the programmer in comparing a string item from the file with another string, there are times when it creates a real inconvenience compared to the cost of the extra disk space involved.

The disadvantage of MF files over FF files is that MF files use more disk space due to the inclusion of the item marker bytes. In this example, the record size is 13% greater than the corresponding FF file in Example 5 though both contain the same data, excepting that the MF file strings are not padded.

```

10 CLEAR 2000
20 OPEN "O",1,"X-XX/DAT:1","MF",71
30 GOSUB 10000      'create data for record
40 IF RN% = 0 THEN CLOSE: END      'no more records
45 IF LOF(1) < RN% THEN PRINT "BAD RECORD #: GOTO 30
50 PUT 1,,,(20)NM$,AN%,AM!,DT#,(15)ST$,IG%,FP!,DP#;      'write record
60 GOTO 30

```

The file is opened for sequential output of records each 71 bytes long. According to the IGEL at line 50, this 71 bytes allows for items of 21, 3, 5, 9, 16, 3, 5 and 9 bytes respectively (remember, each item starts with a marker byte).

The programmer supplies the routine at line 10000 to set RN% = 0 if no more records are to be created. Otherwise it sets RN% non-zero and loads the data for the new record into the 8: variables NM\$, AN%, AM!, DT#, ST\$, IG%, FP! and DP#.

The representation of the items on the disk is the same as described in Example 1 for the MU file excepting that SOR items are not used and that both string items are limited to a maximum number of characters, 20 for NM\$ and 15 for ST\$. If at the time the file item is written, either string variable has a length greater than the maximum allowed for the file item, then the excess characters on the right are not transmitted to the file item.

Strictly speaking, it is not a requirement that string expressions in the IGEL used at line 50 above be prefixed with a maximum string length value. The IGEL of Example 1, line 50 could have been used. However, by not specifying a maximum string length value for any one string item, full update capability cannot be guaranteed for the record.

Note the use of the LOF function at line 45 to check if the requested record is within the file.

Example 14. Randomly read and optionally update records of a MF file.

```
10 CLEAR 2000
20 OPEN "D",1,"XXX/DAT:1","MF",71
30 GOSUB 10000      'determine which record to read
40 IF RN% = 0 THEN CLOSE: END      'end if no more
50 GET 1,RN%,120    'read that record
60 GOSUB 15000      'optionally update the record's data variables
70 IF RN% <> 0 THEN PUT 1,#,120    'if required, rewrite the record
80 GOTO 30
120 (20)NM$,AN%,AM!,DT#,(15)ST$,IG%,FP!,DP#;
```

The file is opened for random reading and writing. The first parameter of the OPEN statement is "D" to prevent extension of the file.

The programmer supplies the routine at line 10000 to determine which record is to be processed next. On exit from that routine, RN% contains the record number except that if RN% = 0, then the run is to end.

The record is then read. The resulting length of the string NM\$ is 0 to 20 characters and of string ST\$ is 0 to 15 characters depending on what the corresponding file item actually had in it. Remember, no padding with spaces was done during the item write and none is done during the GET.

The programmer supplies the routine at line 15000 to query the data in the variables NM\$, AN%, AM!, DT#, ST\$, IG%, FP! and DP#. If the record is not to be updated, it sets RN% = 0. Otherwise, it changes some or all of those variables.

On return from this routine, if RN% is not zero, the record is written back to the file. If one or more of the string variables have a new length, then the corresponding file item assumes that new length.

Example 15. Sequentially write to a MI file.

MI and FI files are one long series of items. If the programmer logically groups items into records, BASIC knows nothing of it since a record length is not specified at OPEN, such as is done for field item, MF and FF files, nor is there a record marker, such as the SOR (start of record) byte for MU files and the EOL (end of line) byte for print/input files. Not knowing anything about the programmer's possible logical record segmentation of MI and FI files, BASIC cannot automatically advance to the next record such as was done by the GET statement at line 80 in Example 12 where the remainder of a type 1 or type 2 record was bypassed.

We will use the code of Example 11 with one change, to generate a MI file consisting of 3 record types (remember, BASIC knows nothing of records in MI and FI files). Changing line 20 of Example 11 to:

```
20 OPEN "O",1,"XXX/DAT:1","MI"
```

we can generate the file used in Example 16 below.

Example 16. Sequentially read a MI file.

The file created in Example 15 is used here. Though the programmer knows the file contains records of 3 types, BASIC does not. Therefore, to advance to the next record, the program must read the previous record completely, though it need not do so all in one GET statement.

A MI file cannot be updated. This restriction is made because of the impossibility of handling strings whose lengths change.

```
10 CLEAR 2000
20 OPEN "I",1,"XXX/DAT:1","MI"
30 IF EOF(1) THEN END 'exit if empty file
40 GET 1,,,RT$; 'read record type character
50 IF RT$ <> "1" THEN PRINT "BAD RECORD TYPE": END
60 GET 1,,,NM$,AN%,AM!,DT#,ST$,IG%,FP!,DP#; 'read type 1 record
70 IF EOF(1) THEN END
80 GET 1,,,RT$ 'read next record type character
90 IF RT$ = "2" THEN 150
100 IF RT$ <> "3" THEN 50
110 GET 1,*,SJ$,DF#,IP%,IA%,FG!; 'read type record
120 GOSUB 11000 'process using type 1 and 3 record data
140 GOTO 70
150 GET 1,,,SA$,SB$,LN$,PD1;: GOTO 70
```

Remember, though the comments in the above program discuss records, the logical segmenting of the file into records is known only to the programmer and not to BASIC.

Note that line 60 above used a null positioning parameter where line 60 in Example 12 used the * positioning parameter. In Example 12, the * was used because file positioning was to stay where it was and not advance to the next record. However, since for MI and FI files, BASIC knows nothing of records, the null and the * positioning parameters work exactly the same which is to leave the file positioned where it is. Thus, in lines 40, 60, 80, 110 and 150 above, the positioning parameters null and * could have been used interchangeably. Since it is easier to type a null than an *, the null will tend to be used. Remember though, for MU, MF and FF file processing, there is a difference between the meaning of null and the meaning of *.

Example 17. Sequentially write records to a FI file and sequentially write index records at the end of that file to indexing into the main records.

The FI file is a very flexible file. It allows the programmer the capability of the FF file while allowing records of different lengths. Remember, as if the MI file, BASIC knows nothing of the programmer's segmenting of a FI file into records. To the programmer though, the FI file can be an assortment of all kinds of records. This example and Example 18 will use a FI file composed of 5 different logical record types: the three record types used in examples 11, 12, 15 and 16, the index records used in the FF file in examples 9 and 10, and another 2 byte record type unique to the current file.

In this example, we will assume the record type 1's AN% item is an account number and that the account number is unique for each type 1 record. The file is first written to contain all the records of the first 3 types. The RBA and the account number of each type 1 record is saved in two BASIC arrays. After all the data records are written, a 1 byte record is written to indicate the end of the data records. Next, the two arrays are sorted into ascending account number order. Index records are then written to the file. Lastly, the number of index records is written to the file.

This example is a cross between Example 9 and Example 11 and most of the comments there apply here, excepting that you are dealing with one FI file instead of a MU file and a FF file.

```

10 CLEAR 2000
20 DIM AN%(4000),RB!(4000)           'arrays for index data
30 OPEN "O",1,"XXX/DAT:1","FI"       'open the combined data/index file
40 RC% = 0
50 RT$ = "1": GOSUB 10000            'create next type 1 record
60 IF RN% = 0 THEN 170               'done with main data
70 RC%=RC%+1: IF RC% > 4000 THEN PRINT "FILE TOO LARGE": GOTO 170
80 RB!(RC%) = LOC(1)!                'RBA where type 1 record will be stored
90 PUT 1,,,(1)RT$,(20)NM$,AN%(RC%),AM!,DT#,(15)ST$,IG%,FP!,DP#; write
type 1 rec
100 GOSUB 15000                      'create type 2 or 3 record
110 IF RT$ = "2" THEN 150
120 IF RT$ <> "3" THEN 50
130 PUT 1,,,(1)RT$,(40)SJ$,DF#,IP%,IA%,FG!; 'write type 3 record
140 GOTO 100
150 PUT 1,,,(1)RT$,(3)SA$,(32)SB$,(14)LN$,PD!; 'write type 2 record
160 GOTO 100
170 IF RC% = 0 THEN PRINT "NO DATA RECORDS": END
175 RT$="0": PUT 1,,,(1)RT$; 'flag end of main data
180 CMD"O",RC%,AN%(1),RB!(1) 'sort index data
190 FOR X = 1 TO RC%
200 PUT 1,,AN%(X),RB!(X); 'write index record
210 NEXT X
220 PUT 1,,RC%; 'write number of index records
230 CLOSE: END

```

An advantage of writing the index records in with the data records is that only one file is used for both, thus avoiding problems in backup and copy of keeping a data and an index file in synchronization with each other. In examples 9 and 10 we could have used only one file, storing the index again on the back end of the MU file.

Since the file is a fixed item type, all named string variables in the IGELs must be prefixed with the length the file item is to have. Truncation or padding with spaces on the right takes place as the string data is moved to the file item.

As with Example 9, exactly the same results would have been attained had text lines 80 and 90 above been written as:

```

80 PUT 1,,, (20)NM$,AN%(RC%),AM!,DT#,(15)ST$,IG%,FP!,DP#
90 RB!(RC%) = LOC(1)# 'RBA where the record was placed

```

Note at line 175 a one byte end-of-main data record is written. This separator is needed by Example 18.

Example 18. Randomly read and optionally update the data records of an indexed MI file.

The file created in Example 17 is used here. The last two bytes of the file are read to determine the number of index records (and type 1 records). The index records are then read into two arrays. Then selected data record groups are read from the file and optionally the DF# and IA% items of the type 3 records are updated back to the file.

This example is a cross between Example 10 and Example 12 excepting that both the index and the data are contained within the FI file, only two items of the type 3 record are updated, and other differences as noted below.

```

10 CLEAR 2000
20 DIM AN%(4000),RB!(4000) 'two arrays for index data
36 OPEN "D",1,"XXX/DAT:1","FI"
40 X! = LOC(1)% - 2 'compute RBA of file's last 2 bytes
50 GET 1,!X!,,RC%; 'read in count of index records records
60 GET 1,!$(X!-6*RC%) 'position file to 1st index record
70 FOR X = 1 TO RC% 'read index data into the two arrays
80 GET 1,,,AN%(X),RB!(X);
90 NEXT X
100 GOSUB 10000 'determine which account to process
110 IF RN% = 0 THEN END 'run is completed
120 FOR X = 1 TO RC% 'search account # array for match
130 IF RN% = AN%(X) THEN 150
140 NEXT X: PRINT "BAD ACCOUNT NUMBER": GOTO 100
150 GET 1,!RB!(X),,(1)RT$,(20)NM$,AN%,AM!,DT#,(15)$,IG%,(12)$; 'type 1 rec
160 IF RT$ <> "1" THEN PRINT "BAD INDEX": END
170 IF RN% <> AN% THEN PRINT "BAD FILE DATA": END
180 GET 1,,, (1)RT$; 'read next record's type char
190 IF RT$ = "3" THEN 230
195 IF RT$ = "1" OR RT$ = "0" THEN 100 'test end of account
200 IF RT$ <> "2" THEN PRINT "BAD FILE DATA": END
210 GET 1,*,,(3)SA$,(32)SB$,(14)LN$,PD!; 'bypass type 2 record
220 GOTO 180
230 GET 1,,, (40)SJ$,DF#,(2)$,IA%,FG!; 'read type 3 record
240 GOSUB 11000 'process type 1 and 3 record data
250 IF RN% <> 0 THEN PUT 1,$,,(40)$,DF#,(2)$,IA%,(4)$; 'update type 3 rec
260 GOTO 180

```

At line 40, note the use of the LOC(1)% function to obtain the RBA of the file EOF. This is used to compute the RBA of the next to the last byte in the file as the files last 2 bytes contain the integer count of both the number of index records in the file and the number of type 1 records in the file. This integer value is read into RC% at line 50.

At line 60, using this index record count, the RBA of the 1st index record is computed and the file positioned to the start of the first index record. Note that the computation is done right in the GET statement's file positioning parameter. This may be done provided the computation itself does not reference a filearea. Also note that the file positioning parameter is of the !\$rba type (see section 8.8.6), meaning that this GET is for file positioning only; an IGEL or IGELSN is not allowed, and no data transfer takes place.

The programmer supplied routine at line 10000 determines the account number for the group of records to be interrogated. If no more accounts are to be read, set RN% = 0; otherwise set RN% = to the account number.

At line 150 the file is positioned using the RBA associated with the account number in the index arrays. The type 1 record is then read. It has been arbitrarily decided that it is not necessary to know what is in the file items corresponding to ST\$, FP! and DP# (see line 90 in Example 17). Therefore, these items can be bypassed. However, since this is fixed item file, it is necessary to inform BASIC of the number of file bytes to be skipped, using the (len)\$ format (see section 8.4.3.3). The (15)\$ causes the skip of the 15 file bytes that would normally be read into the string ST\$. The (12)\$ causes the skip of the 4 bytes that would normally be read into FP! and the 8 bytes that would normally be read into DP#. Lastly, if this IGEL was used in FF file processing, the (12)\$ expression could have been dropped from the IGEL, as no other expressions follow it in the IGEL and the next GET with null positioning parameter will advance the file to the next record. In FI processing, the programmer must account for all of the record's space since BASIC knows nothing of his/her record structure, hence the (12)\$ is required.

At line 210, though all we want to do is skip the type 2 record, we still must advance the file positioning as once again, BASIC knows not where this record ends. The same file position advancement could have been obtained with:

```
120 GET 1,,,(53)$;      'bypass rest of type 2 record
```

The programmer supplied routine at 11000 processes the data from the type 1 and 3 records. If the type 3 record is not to be updated, set RN% = 0. Otherwise change either or both of DF# and IA% and set RN% non-zero.

The PUT statement at line 250 repositions the file to the REMBA position remembered by the GET at line 230. The type 3 record is then updated. Note that only 2 of the items, those corresponding to DF# and IAA, are replaced in the file (compare with line 130 of Example 12). The other items are skipped over and are not changed. In using the (len)\$ or (lend expressions in the IGEL, the programmer must be certain to account for the proper number of bytes.

The use of the (len)\$ type expression in the IGELs of lines 150 and 250 was done only to give examples of the (len)\$ use. The user might prefer to just use the regular IGELs, changing those two lines to be:

```
150 GET 1,*,,(1)RT$,(20)NM$,AN$,AM!,DT#,(15)ST$,IG$,FP!,DP#;
and
250 IF RN% <> 0 THEN PUT 1,$,,(40)SJ$,DF#,IP$,IA$,FG!;
```

APPENDIX C

NEWDOS/80 VERSION 2.5 --- THE NEWDOS/80 VERSION 2 HARD DISK SYSTEM

- Section 1: Overview
- Section 2: Comments and Restrictions
- Section 3: Changes to PDRIVE
- Section 4: Formatting your hard disk
- Section 5: Moving NEWDOS/80 to the hard disk.
- Section 6: Defining PDRIVE slots from a volume definition file
- Section 7: Backing up hard disk to diskettes

The NEWDOS/80 Version 2 modified for hard disk operations is called both the NEWDOS/80 Version 2 Hard Disk Operating System and NEWDOS/80 Version 2.5. The difference between the regular NEWDOS/80 Version 2 and Version 2.5 is the inclusion in 2.5, via patches, of code to handle the hard disk. This documentation for NEWDOS/80 Version 2.5 is considered as Appendix C to the regular NEWDOS/80 Version 2 manual and should be inserted into that manual after Appendix B.

NEWDOS/80 Version 2.5 is NOT offered as a stand alone DOS; the regular NEWDOS/80 Version 2 must be PURCHASED and REGISTERED either prior to or at the time of purchase of NEWDOS/80 Version 2.5.

As usual with NEWDOS/80, the user should study this document carefully before attempting to do anything with the NEWDOS/80 Version 2.5 Hard Disk Operating System.

The basic NEWDOS/80 Version 2.5 supports Apparat's and Tandy's hard disks for either the TRS-80 Model I or III. Special Version 2.5 system diskettes may later be made available for other types of hard disks.

You may have already been using your hard disk under LDOS or another DOS and have valued user files on the hard disk. If this is so, you are already a serious hard disk user and cannot afford to lose valued data just because you are switching DOSs. You Faust:

CAREFULLY plan your move.

Backup up those files from hard disk to diskette using that DOS's offload program. This is insurance in case the conversion to NEWDOS/80 fails; you can reformat the hard disk(s) for the other DOS and reload the files.

Move NEWDOS/80 Version 2.5's HDBACKUP/CND program over to that DOS (see the NND parameter discussion in section 7).

Use HDBACKUP under that DOS to again offload your files to another set of diskettes. Remember to use the SAVE, INCLUDE and NND parameters. Also remember, for 5 million bytes of data, the HDBACKUP SAVE function will need 27 pre-formatted single sided, double density 40 track (720 sector) diskettes.

Initialize the hard disks for NEWDOS/80 Version 2.5 using HDFMTAPP, PDRIVE and FORMAT.

Use HDBACKUP under NEWDOS/80 Version 2.5 to RESTORE the files from diskettes to the hard disks. Carefully plan this move; you may decide to use more than one RESTORE from the same backup to get the various files where you want them.

1. OVERVIEW

1. A data file may contain as many as 16 million bytes.
2. A hard disk data volume can be up to 65535 sectors and contain up to 246 user files.
3. PDRIVE allows a maximum of eight active slots with a maximum of 4 floppy data volumes or eight hard disk volumes active at one time.
4. The capacity to support hard disk drives of over 100 million bytes exists, though currently only the Apparat and Tandy hard disk drives are supported.
5. A hard disk drive is divided into one or more drive sections.
6. A hard disk section is divided into one or more data volumes. The data volume is what is defined by PDRIVE. A data volume may not span multiple hard disk drives or drive sections.
7. 48K of RAM is required. The hard disk modifications for NEWDOS/80 Version 2.5 have preempted computer main memory 0F900H - 0FFFFH. Programs that execute in that area must no longer be used.
8. Aside from the main memory limitation above, most programs that work with NEWDOS/80 Version 2 will work with NEWDOS/80 Version 2.5. However, if any program assumes certain volume sizes (i.e. 350 sectors on the Model I or 720 sectors on the Model III) or a certain location and size of the directory, that program will have to be modified. Basically, programs that use standard file I/O and observe HIMEM should be OK.
9. The hard disk system can operate either from floppy drive 0 (in which case floppies retain their old drive numbers) or from the hard disk (in which case, floppy drives 0 - 3 become drives 4 - 7 respectively). Section 5 steps the user through the shift of the system from a floppy to a hard disk volume.
10. Program HDFMTAPP/CMD is used to magnetically format Apparat and Tandy hard disks.
11. Program HDBACKUP/CMD is used to selectively save hard disk files of any size onto floppies or to selectively restore them to hard disk. This is NEWDOS/80 Version 2.5's hard disk backup facility and must be used to make backup copies of valued data files. Further, the program HDBACKUP/CMD can be transferred to another DOS (see NND parameter in section 7) so that the program can be used under that DOS to offload user files to diskette preparatory to changing the hard disk to operate under NEWDOS/80 Version 2.5 on to onload user files from diskette to hard disk should the user wish to take the hard disk back to the other DOS.

12. Program EXTPDRIV/BAS can be used to set PDRIVE slot definitions from definitions stored in an ASCII text file. Since hard disk data volume specifications are both difficult and critical, it is recommended they be permanently built in a text file via SCRIPSIT or CHAINBLD and then activated when needed via EXTPDRIV.

13. Parameter HDS has been added to DOS command PDRIVE to define hard disk volumes.

14. Hard disk volumes defined under Model III NEWDOS/80 can be used under Model I NEWDOS/80 and vice versa. The files on these volumes are NOT useable interchangeably if they were NOT useable interchangeably when those files were on diskettes.

2. COMMENTS and RESTRICTIONS:

1. The user must be knowledgeable of NEWDOS/80 Version 2 and all subsequent information issued via the zaps prior to attempting to use the Hard Disk system. All hard disk discussion herein assumes this knowledge. This document is intended only as supplementary information to the regular NEWDOS/80 Version 2 manual and its subsequent zaps.

2. This document does NOT provide information about your hard disk. That information must be obtained from the source where you purchased or otherwise obtained your hard disk. The information NEWDOS/80 needs to know about your hard disk drive is (1) the number of recording surfaces (or number of I/O heads), (2) the number of tracks per surface (or the number of cylinders), (3) the number of 256 bytes sectors actually formatted on each track, and (4) track-to-track stepping rate code.

3. NEWDOS/80 Version 2 was not designed to operate with hard disks. All of the changes creating Version 2.5 have been done by patching the standard Version 2, with the exception that SYS0/SYS has been extended five sectors. This patching to Version 2 provides a minimum hard disk operating system and each specially purchased hard disk system diskette will operate with one and only one type of hard disk drive. If another type of hard disk drive has exactly the same interface to the computer, then it can also work with a particular hard disk system diskette (example, both Tandy's and Apparat's hard disks for the Model I and Model 3 have the same software interface, therefore either (but not both at the same time) can be used with the same Version 2.5 Hard Disk system diskette). The standard issue NEWDOS/80 Version 2.5 hard disk system supports Apparat's and Tandy's hard disks for either the TRS-80 Model I or III.

4. This system REQUIRES 48K of RAM. To implement hard disk code, main memory from F900H to FFFFH has been taken by DOS and is not available to the users. Any user programs that use this area MUST either no longer be used or be modified to no longer use the F900H to FFFFH main memory area. HIMEM is set to 0F8FFH by DOS automatically and programs that observe HIMEM should be all right.

5. The number of active PDRIVE slots (formerly called drives) has been expanded from 4 to 8, allowing a maximum of 4 floppy volumes, 8 hard disk volumes or a combination thereof. The number of actual active

PDRIVE slots is still controlled by SYSTEM option AL. If a ?DRIVE active slot is to be unused, defining it as a hard disk volume and setting the PDRIVE HDS sub-parameter vscl to 0 will cause PDRIVE to accept the definition in an active slot and NEWDOS/80 to treat the slot as DEVICE NOT AVAILABLE whenever it attempts to use that slot (drive).

6. Two floppy drives are desirable, though only one is required. The Version 2.5 Hard Disk System comes on a standard 40 track, double density diskette on the Model III and a standard 35 track, single density diskette on the Model I. Since the capacity of the Model I diskette is too small to contain all the files for the hard disk system as well as those of the nor.-hard disk system, some of the files (or program modules) of the regular NEWDOS/80 are not present on the hard disk system diskette. When needed, you may copy these modules over from the regular NEWDOS/80 system diskette.

7. The user may elect to run using a floppy system diskette (a copy of the Version 2.5 Hard Disk System diskette) or he/she may move the NEWDOS/80 Version 2.5 Hard Disk System onto a hard disk volume.

1. If the system is being run from floppy drive 0 (the normal drive 0 for the computer), the floppy drives 0 - 3 use PDRIVE slots 0 through 3 respectively, just as they do in the regular NEWDOS/80 Version 2. Under the floppy hard disk system, PDRIVE slots 1 - 7 may be defined as hard disk volumes and accessed by user programs as drives 1 - 7 respectively.

2. If the system is being run from hard disk, the floppy drives 0 - 3 use PDRIVE slots 4 through 7 respectively, and the floppy drives 0 - 3 are known to the system and user programs as drives (or slots) 4 through 7 (though you may use any or all of drives (or slots) 0 through 7 for hard disk volumes).

***** Warning, when the system is being run from the hard disk, access to the floppy drives is slots 4 - 7 meaning that all slots between the system volume in slot 0 and the slot being used by the floppy drive MUST be valid definitions even though you are using only one hard disk volume. PDRIVE will allow and DOS will ignore a slot defined for a null hard disk volume (having the HDS sub-parameter vscl equal 0), thus allowing access to the floppy drives.

8. The DOS command FORMAT or the format portion of COPY do NOT actually format the hard disk; instead of formatting, the message, INITIALIZING SECTORS, is displayed and the sectors are written with a standard pattern. To actually format the Apparat or Tandy hard disk, use the HDFMTAPP/CAD program provided. To format another hard disk, you must use a program provided by the hard disk retailer (NOT provided by Apparat).

9. This hard disk upgrade does NOT support the standard LDOS TRS-80 hard disk data volumes as directory concepts slightly differ, though those volumes can be read via SUPERZAP by expert users (provided the hard disk is divided into sections properly, spgl value is 16, the ddsll value is 76, the ddsal value is 32, and the gp11 value is 2 for one surface volumes, 4 for two surfaces, 6 for three surfaces and 8 for four surfaces)(changing HIT sector rel byte 1FH from 00H to 16H allows DIR to work and many other functions marginally)(you are on your own processing LDOS volumes under NEWDOS; don't call Apparat when you get into trouble). Basically, when shifting from one DOS to another, the user must off-load the hard disk files to floppies under that DOS using the NEWDOS/80's HDBACKUP program and bring them back in under the other DOS using NEWDOS/80's HDBACKUP program (the NND parameter must be used when the DOS is other than NEWDOS/80).

10. A number of user programs read and interpret the directory. If that program was reading the directory as the DIR/SYS file, observing the protected sector error code and observing EOF, there should be no problem. If the program was using the DDSL value in the data volume 1st sector to compute the directory location, the program will fail unless the data volume has spgl = 5. If the program was assuming the location and size of the directory, it will most probably fail!!!

11. A data volume must not exceed 65535 sectors. Aside from the space used by BOOT/SYS and DIR/SYS on that data volume, all the remaining space may be allocated to one file, over 16 million bytes. The sector range assigned to one data volume must NOT overlap that of any other data volume; it is the user's responsibility, through careful PDRIVE definition of the data volumes, to avoid this overlap, which can be quite disastrous.

12. A hard disk drive is logically divided into one or more data volumes via judicious use of the PDRIVE HDS parameter. Though a data volume is limited to a maximum of 65535 sectors, a hard disk drive is limited ONLY by its actual capacity AND the limitations that Sectors Per Track (SPT or spgl) must be less than 256, Tracks Per Cylinder (TPC)(or RSC (Recording Surface Count)) must be less than 256, Tracks Per Surface UPS or tps1) must be less than 65536, and TPS times TPC must be less than 65536.

13. A hard disk physical drive's space may be divided into drive sections. Normal NEWDOS/80 Version 2.5 operations DO NOT require this. However, if your division of the hard disk is to be such that part of the hard disk is to be used for data volumes of another DOS (such as LDOS) which assigns data volumes in units of one or more entire recording surfaces, it is necessary to sectionalize your hard disk under NEWDOS/80. This is done by setting the PDRIVE HDS sub-parameter sscl value to the number of recording surfaces assigned to that drive section and by setting, the sfsl value to the relative number of the first recording surface assigned to that drive section. For a given drive, no two sections may share the same recording surface, and no data volume may have space assigned from more than one drive section.

14. For DOS command COPY, the =tc1 parameter is not legal if the SOURCE is a hard disk data volume. For FORMAT and COPY, the =tc2, DDSL and DDGA parameters are not legal if the DESTINATION is a hard disk

data volume. For FREE and the header of DIR, to avoid ambiguity, a track count of 0 is displayed if the data volume is on a hard disk.

15. The SUPERZAP displays may look awkward as they were not designed to handle over 9999 sectors. However, they do work, excepting that TRK and SOT values are not displayed for sectors on hard disk. The DTS main menu function is not allowed for hard disk volumes.

16. Format 5 COPY (full diskette COPY) requires that SOURCE and DESTINATION have the same GPL and SPG values and, if the destination is on a hard disk, the same dds11 and ddsal value. Otherwise format 6 COPY (Copy By File) must be used.

17. Hard disk volumes defined under Model III NEWDOS/80 can be used under Model I NEWDOS/80 and vice versa if NEWDOS/80 supports the drive for the Model I and III. The files on these volumes are NOT useable interchangeably if they were NOT useable interchangeably when those files were on diskettes (such as system program and most user, non-BASIC program files). If you intend to use a hard disk with both your Model I and your Model III (though not at the same time) and intend to run the system from that hard disk, you should create two system volumes on the hard disk, one for the Model I and one for the Model III.

18. ***** Errors may occur in DIRCHECK and SUPERZAP if DFG (MINI-DOS) or 123 (DEBUG) are used during the program's execution and the target drive is not explicitly re-specified after conclusion of MINI-DOS or DEBUG. After MINI-DOS or DEBUG in SUPERZAP, it is recommended that you return to the main menu or do the 'J' display function; for DIRCHECK, respond Y or N to the menu.

3. CHANGES TO PDRIVE for hard disk operation.

No existing parameters in PDRIVE have been changed (so floppies are defined exactly as before), and one parameter, the HDS parameter, has been added to accommodate the hard disks.

The TRS-80 diskette directory was originally intended for 35 or 40 track diskettes of 350 to 400 sectors. In NEWDOS/80 Versions 1 and 2, the directory was modified somewhat to allow for a maximum of 222 user files instead of 62 and allow a maximum of 1536 granules instead of 192. To get these extra granules, the granule lockout table was eliminated from the GAT sector and number of granules per lump (GPL) was expanded from the old implied value of 2 to a user specified value with a maximum value of 8. At 5 sectors per granule, this allowed for 7680 sectors (1,966,080 bytes) per data volume.

However, with hard disks, we really want the capability of allowing a volume to be up to 65535 sectors and a file to be not much less than that. In order to retain the same directory structure but increase the number of sectors for a data volume, we have changed the number of Sectors Per Granule from the old implied value of 5 to a user specified value of not more than 255. Theoretically, this should allow for $1536 * 255 = 391,680$ sectors, but there is another governing restraint, that of the NEXT and EOF fields of the directory FPDE and the file's FCB. These fields allow for a maximum of 65535 sectors (if wrap around is to be avoided). Normally this restriction limits the size of a file, but actually this restriction limits a data volume's size

since NEWDOS/80 has a special use of the FCB that allows sector I/O directly to a data volume, bypassing the file concept altogether. Therefore, the NEWDOS/80 version 2.5 hard disk system limits a data volume to 65535 sectors (16,776,960 bytes). Since each volume has a BOOT/SYS file and a DIR/SYS file, the maximum size of a user file is somewhat less than 65535 sectors.

Though 5, 10 or 15 million byte hard disk can be treated by NEWDOS/80 as one data volume, it is generally desirable to divide a hard disk into more than one data volume. NEWDOS/80 allows the user great flexibility in this, admittedly at a cost of complexity (as usual with NEWDOS/80's PDRIVE which many users are still uncomfortable with). A PDRIVE slot definition actually specifies a data volume, not a floppy drive or a hard disk drive or a hard disk drive section. The specifications for the drive and, optionally, drive section are simply part of the specifications of a data volume.

The definition of hard disk data volumes is more difficult and more critical than for floppy diskette data volumes. The user is solely responsible of assuring that a hard disk sector is NOT shared by two or more data volumes. As an aid to the user, the BASIC program EXTPDRIV/BAS has been provided to search an ASCII text file for a specified definition and assign the definition to a specified PDRIVE slot. Using SCRIPSIT or CHAINBLD, the user can carefully and permanently build his/her hard disk data volume definitions (actually just the HDS parameters), and later, when a particular data volume is needed in a particular PDRIVE slot, EXTPDRIV can be used to effect this assignment.

Further, NEWDOS/30 Version 2.5 does NOT maintain a table of bad hard disk sectors. If your hard disk has bad sectors, you must either operate that drive with a sufficiently reduced SPT (sectors per track) value or you must define data volumes such that the bad sectors are not included within any data volume.

Since a hard disk data volume's definition has more values than for a floppy data volume, and we want to limit each slot's definition to one line on the display, we have decided to combine all 12 values of a hard disk data volume specification into one parameter, the SIDS (Hard Disk Specification) parameter. The 12 values are called sub-parameters; ALL 12 MUST be given EACH time the HDS parameter is used, and all must be in the exact order specified. The specification of the LIDS parameter is:

```
HDS=(hddn1,tps1,sfsl,sscl,spt1,tsrl,vfsl,vscl,spgl,gpl1,ddsl1,ddsal)
```

where:

1. hddn1 means Hard Disk Drive Number and is the relative number (0 3) of the drive on the hard disk cable with 0 being the first drive. hddn1 specifies which physical hard disk drive the data volume is on.
2. tps1 means Tracks Per Surface and is the number of tracks per recording surface (also the number of cylinders) for the hard disk drive. Each recording surface of the drive has tps1 number of tracks. tps1 is an integer from 0 to 65536. For Apparat hard disks, tps1 = 306. For Tandy 5 Meg hard disks, tps1 = 153.

LDOS 5.1.3 appears unable to support the tps1 value of 306 used with Apparat's hard disk. However, an Apparat 10 Meg hard disk (with tps1 =

306 and RSC = 4) can be used as a 5 Meg hard disk with LDOS 5.1.3 where implied values of tps1 = 153 and RSC = 4 are used.

3. sfs1 means Section First Surface and is the relative number of the first surface of the hard disk drive assigned to the drive section containing the data volume. sfs1 is an integer between 0 and RSC-1. If you are not sectioning your hard disks, sfs1 will always be 0.

RSC means Recording Surface Count and is the number of recording surfaces for the hard disk. Another term for the number of recording surfaces is TPC (Tracks Per Cylinder). For Apparat 5, 10 and 15 Meg hard disks, CSC is 2, 4 and 6 respectively. For Tandy 5 Meg hard disks, RSC is 4.

4. sscl means Section Surface Count and is the number of consecutive surfaces of the hard disk drive assigned to the drive section containing the data volume. sscl is an integer between 1 and RSC with the sum of sfs1 and sscl not greater than RSC. If you are not sectioning your hard disks, sscl will always equal RSC.

5. spt1 means Sectors Per Track and is the number of 256 byte sectors on each track of the hard disk drive which in turn is the number of sectors formatted on each track by the format program supplied with your hard disk for Apparat and Tandy hard disks, this is the HDFMTAPP program). spt1 is an integer from 1 to 255. Normally, Apparat and Tandy hard disk drives have 32 sectors per track; however, if during HDFMTAPP formatting of the hard disk, a track is found with more than one error sector, it will be necessary to format the hard disk with less than 32 sectors per track unless you intend to define data volumes to bypass the bad sectors; remember, NEWDOS/80 does NOT maintain a hard disk bad sector table.

6. tsrl means Track Stepping Rate and is a code used by DOS to send track-to-track stepping rate information to the hard disk controller when it is necessary to move the disk arm which contains the read/write heads. tsrl is an integer between 0 and 255. Apparat hard disk use tsrl = 0. Tandy 5 Meg hard disks require tsrl = 6.

7. vfs1 means Volume First Sector and is the relative sector number within the drive section of the data volume's first sector (the data volume's relative sector 0). vfs1 is an integer between 0 and 16,777,215 with an effective upper limit of one less than the number of sectors assigned to the drive section (if a hard disk is not sectioned, the hard disk is one in the same as its one section). If vfs1 = 0, then the data volume's sector range starts with the first sector of the drive section; further, if both vfs1 and sfs1 are 0, the data volume's range starts with the drive's 1st sector.

8. vscl means Volume Sector Count and is the number of consecutive sectors of the drive section, beginning with sector vfs1, assigned to this data volume. vscl is an integer between 0 and 65535 but the sum of vfs1 and vscl must not exceed the number of sectors assigned to the drive section (which is tps1 * sscl * spt1). If vscl is simply the asterisk character instead of an integer, PDRIVE will assign all of the drive section's remaining sectors to the data volume.

***** IMPORTANT. PDRIVE will accept a vscl value of 0, meaning a null data volume, and it will allow the data volume definition into an active slot (provided the definition has no other errors). If vscl is 0, DOS will generate a DEVICE NOT AVAILABLE error whenever a slot is selected that contains this data volume. This is needed as a way of filling in PDRIVE slot definitions so that access can be made to slots 4 - 7 for floppy diskette operations when running the system from hard disk and not all of slots 1 to 3 are defined for valid hard disk data volumes.

The sub-parameters hddn1, tps1, sfs1, sscl, spt1, vfs1 and vscl combine to define a unique range of hard disk sectors assigned to the data volume. No sector in this range may be shared by another data volume defined by PDRIVE; it is the user's responsibility to avoid this conflict. Otherwise, the same sector can end up being used for two different purposes.

9. spg1 means Sectors Per Granule and is the number of sectors in each allocation granule for this data volume. spg1 is an integer between 1 and 255. If spg1 = 0 is specified, PDRIVE will compute the lowest spg1 above 4 that will suffice for the gpl1 value specified and the number of sectors assigned to the data volume (vscl).

When DOS assigns disk space to a file, it does so in minimum units called granules; so the spg1 value defines the minimum number of sectors allocated to a file and also is one more than the maximum number of sectors that a file will have allocated beyond its needs. Generally, it is desirable to have a small spg1 value, but the smaller the spg1 value, the smaller the maximum size a data volume may be. In the regular NEWDOS/80 Version 2, a SPG value of 5 was implied and always used, except in some of the COPYs to and from special TRSDOS diskettes. If full diskette COPY (not CBF) compatibility is wanted with the floppies, spg1 = 5 must be used as that is the standard in the NEWDOS/80 Version 2 floppy world.

***** Warning, when a NEWDOS/80 system volume is being COPY'ed using CBF and the destination spg1 value is less than the source spg1 value, DISKETTE GAT OVERFLOW error may occur. The only alternative is to copy the system from the hard disk system diskette and use a destination spg1 greater than 4.

10. gpl2 means Granules Per Lump and is the maximum number of allocation granules for each byte in the data volume directory's Granule Allocation Table in the GAT sector (the first sector of the directory). gpl1 is an integer between 2 and 8. GPL = 2 is the standard for the old Model I TRSDOS 2.3, and the NEWDOS/80 Version 2 master diskettes use GPL = 2. However, any data volume, whether hard disk or floppy, with more than 1920 sectors, should use a larger GPL under the criteria that it is better to increase the GPL than the SPG. It is recommended that if GPL = 2 is not used, then use GPL = 8. Though the other values are legal, don't use them unless you are attempting compatibility with another DOS.

A lump???? For NEWDOS/80 Version 2, we wanted to eliminate the one-to-one correspondence between a byte in the GAT table and a diskette (or hard disk) track or cylinder so that granules could flow across track and cylinder boundaries. A granule's allocation state is

handled by one bit in the GAT, and we wanted to use all eight bits in each GAT byte to extend the number of granules the GAT could account for. However, the old TRSDOS 2.3 standard was to use only the right two bits of each GAT byte; so we couldn't arbitrarily force all directories to start using all 8 bits. Yet, we wanted to allow use of all 8 bits; so we had to come up with a name for a byte in the GAT as distinct from anything else. Under the assumption that if a number of sectors is a granule, then a number of granules could be called a lump, we defined a lump to be simply a byte in the Granule Allocation Table in the data volume directory's first sector, and that's all it is.

11. `ddsl1` means Default Directory Starting Lump and means the relative number of the lump whose 1st sector is the beginning of the data volume's directory. `ddsl1` is an integer between 1 and 191, though no guarantee is given that a particular value will work. The standard `ddsl1` value in the 35/40 track single sided, single density diskette world was and is 17, and your master NEWDOS/80 system diskette uses that value. If `ddsl1 = 0` is specified, NEWDOS/80 will compute a `ddsl1` value somewhere near the middle of the data volume, but not greater than 80, as it is assumed more data will exist near the beginning of the volume than at the end.

All Model I and Model III DOSS put the directory somewhere in the middle of the data volume. Since NEWDOS/80 runs with a variety of diskette and hard disk capacities, NEWDOS/80 allows the user to specify where the directory is to be put. The `ddsl1` value is this specification. NEWDOS/80 stores the `ddsl1` value in 3rd byte of the first sector of BOOT/SYS (also the first sector of the data volume) during data volume format (either FORMAT or COPY) so that DOS (and clever users) can find the directory. NEWDOS/80 senses it has lost the directory location when it reads a directory sector that is not protected. It then goes to the 3rd byte of the volume's 1st sector for the `ddsl1` value and computes the directory location. The standard DDSL value for the 35 and 40 track single density diskettes was 17, but as diskettes have increased in capacity and hard disks have appeared, starting the directory at lump 17 placed it too close to the start of the data volume. For dual sided 80 track, double density diskettes with GPL=8, it was common to put the directory at lump 35.

In the diskette world, DDSL has meaning only when a diskette is formatted as NEWDOS/80, at all other times, can find the directory when it wants to. However, in the hard disk world, since we can't write directory sectors with address marks different from the other sectors, NEWDOS/80 cannot tell when it should go to the volume's first sector, get the `ddsl1` value, and re-compute the location of the directory. Therefore, the `ddsl1` value is used by NEWDOS/80 at all times to know where a hard disk volume directory is. If you change the `ddsl1` value at a time other than just before the hard disk volume is formatted, NEWDOS/80, without realizing it, will process non-directory data as directory data.

12. `ddsal` means Default Directory Sector Allocation and specifies the number of sectors to be used for the directory. `ddsal` is an integer from 10 to 33. This `ddsal` value is different than the DDGA value used by PDRIVE for floppy diskette definitions. Do not confuse the two. The change from DDGA to DDSA was necessitated by the fact that SPG for the hard disks is no longer a standard 5 sectors per granule. A `ddsal` value

of 10, 15, 20, 25 and 30 is compatible with older configurations that used DDGA=2, 3, 4, 5 or 6 respectively. A ddsal value of 33 allows a data volume to have a maximum of 246 user files. Unless a hard disk data volume is to be small or compatibility with diskettes is to be maintained, it is recommended that ddsal = 33 be used.

The ddsal value for hard disk is more important than the DDGA value is for floppies. The DDGA value is used only at diskette format time. The ddsal value, along with the ddsll value, is the only way the DOS sector I/O routines know if a sector is part of the hard disk data volume directory or not; therefore, if the ddsal value is to be changed, it must be changed only before a hard disk data volume is formatted. The ddsll and ddsal values are the ONLY way the NEWDOS/80 sector I/O routines know that a given hard disk sector is a directory sector.

EXAMPLES:

***** Remember, when parameter HDS is specified, all 12 sub-parameters must be supplied in the correct order.

1. PDRIVE,0,1,HDS=(0,306,0,2,32,0,0,2880,5,8,35,33)
specifies a 2880 sector data volume with 5 sectors per Granule, 8 granules per lump, a 33 sector directory positioned at the start of lump 35. TLC first 2880 sectors of the first drive section of hard disk drive 0 will be allocated to this volume. The drive section consists of the first 2 recording surfaces of the drive, which may or may not be all that the drive has. Each recording surface has 306 tracks. Each track has 32 sectors and the drive's stepping rate code is 0. This data volume can be accessed by user programs as drive 1.

2. PDRIVE,0,2,HDS=(1,153,1,3,32,6,2000,10000,0,8,0,33)
specifies a data volume on hard disk drive 1 that has 153 tracks per surface and 32 sectors per track. The drive section consists of the 2nd, 3rd and 4th recording surfaces. The data volume consists of 10,000 sectors beginning with the drive section's relative sector 2,000. PDRIVE will compute the sectors per granule and use 8 granules per lump. PDRIVE will compute the position of the 33 sector directory within the volume. User programs will access this data volume as drive 2.

3. PDRIVE,0,1,HDS=(0,153,0,4,32,6,0,*,0,8,0,33) specifies a data volume that occupies all 19,584 sectors of the first four recording surfaces of hard disk drive 0. The vscl, spgl and ddsll values are computed by PDRIVE. User programs will access this data volume as drive 1.

4. PDRIVE,0,3,HDS=(0,153,0,1,32,6,0,0,5,2,17,33)
specifies a null data volume (vscl value is 0). NOTE, all other sub-parameters must be valid. If a user program attempts I/O via drive 3, DEVICE NOT AVAILABLE, error will occur. However, the FREE command and any other DOS functions that search the various drives will ignore drive 3.

EXAMPLES OF PDRIVE COMBINATIONS:

1. Settings to exactly overlay the standard LDOS values on a single Tandy 5 Meg drive where each of 4 volumes has one surface.

```

HDS=(0,153,0,1,32,6,0,4896,5,8,61,33)
HDS=(0,153,1,1,32,6,0,4896,5,8,61,33)
HDS=(0,153,2,1,32,6,0,4896,5,8,61,33)
IDS=(0,153,3,1,32,6,0,4896,5,5,61,33)

```

This divides the hard disk drive into 4 drive sections, each containing one data volume. If you assign the 4 definitions to PDRIVE slots 0 - 3 respectively, you must have moved the NEWDOS/80 system to hard disk as described in section 5. However, if you assign these definitions to slots 4 - 7 and have previous file data from LDOS operation, you can look at that data via SUPERZAP (if you are interested), and you can look at the directory starting at relative sector 2432.

2. The user has one Apparat 5 Meg drive, fundamentally wants all his/her user files accessible via drive 1 with a small amount of work space on drive 2. The user wants to run using a hard disk system volume for drive 0 and to be able to access to his two floppies via slots 4 and 5. With SYSTEM option AL = 6, the definitions for slots 0 - 5 will be as follows:

```

HDS=(0,306,0,2,32,6,6,720,5,8,17,10)
HDS=(0,306,0,2,32,6,720,16864,11,8,80,33)
HDS=(0,306,0,2,32,6,17584,2000,5,8,25,33)
HDS=(0,1,0,1,1,0,0,0,5,8,17,10) a dummy definition
TI=A,TD=E,TC=40,SPT=18,TSR=0,GPL=2,DDSL=17,DDGA=2
TI=A,TD=E,TC=40,SPT=18,TSR=0,CPL=2,DDSL=17,DDGA=2

```

Note that the 8th sub-paraneter (vscl) of HDS is the number of sectors assigned to the data volume (NOT the ending sector number). Slot 3 has been defined as a dummy (the vscl value = 0) to allow FREE to get to slots 4 and 5.

3. The user has two Apparat 10 Meg drives and wants the system volume on hard disk, 3 hard disk data volumes with slot 1 to contain all the space of the 2nd drive. The definitions for slots 0 - 7 could be:

```

HDS=(0,306,0,4,32,0,0,5595,5,8,69,33)
HDS=(1,306,0,4,32,0,0,39168,26,8,94,33)
HDS=(0,306,0,4,12,0,5595,5595,5,8,69,33)
HDS=(0,306,0,4,32,0,11190,5595,5,8,69,33)
HDS=(0,306,0,4,32,0,16785,5595,5,8,11,33)
HDS=(0,306,0,4,32,0,22380,5595,5,8,69,33)
HDS=(0,306,0,4,32,0,27975,5595,5,8,69,33)
HDS=(0,306,0,4,32,0,33570,5595,5,8,69,33)

```

4. FORMATTING YOUR HARD DISKS.

Hard disks must be formatted before they can be used with NEWDOS/80 Version 2.5 or any other DOS. Some hard disk manufacturers format their hard disks before shipping the drive and have internal coding to bypass error sectors automatically, and if this is the case, you may bypass this section on hard disk formatting.

NEWDOS/80 Version 2.5 does not maintain an error sector table and assumes the consecutive sectors that it can read from a hard disk are error free. Bad (error) sectors must be hidden from NEWDOS/80. One way to do this is to reduce the number of data sectors per track, allowing HDFMTAPP to write a dummy sector over the bad spot on the track. Another way is to later define (via PDRIVE) the data volumes such that the bad sectors are not part of any data volume.

NEWDOS/80 DOS commands FORMAT or COPY with format do not actually format a hard disk. The actual formatting must be done either by a stand alone program or by a program that operates under NEWDOS/80 but does all of its own I/O to the hard disk. NEWDOS/80 Version 2.5 provides the program HDFMTAPP to format Apparat's and Tandy's hard disks for the Model I or III. The format program for other types of hard disk drives must be supplied to the user by that hard disk drive retailer.

Formatting a hard disk destroys all information on that hard disk. If you must re-format a hard disk, be sure to extract as much valued information from that hard disk (you may use program HDBACKUP) as you can before re-formatting.

Though we recommend that you format the hard disk drive before use with NEWDOS/80 so that you will be made aware of all the error sectors, a previous format done for another DOS (such as done during the LDOS 5.1.3 hard disk initialization) can suffice if there were no error sectors or you know where they are for bypassing in your definition of data volumes using PDRIVE, and if you know the parameters needed for PDRIVE's HDS parameter. If you elect to do this, then bypass the rest of this section (on HDFMTAPP). An example where you might want to do this is where you wish to share the hard disk between one or more existing LDOS volumes and one or more NEWDOS/80 volumes, thus allowing both LDOS and NEWDOS/80 to use the hard disk (though not both at the same time and not the same data volumes).

To format an Apparat or Tandy Model I or III hard disk, assure the hard disk drive is properly connected to the computer and power is on; then execute the DOS command HDFMTAPP, proceeding as follows:

1. Reply the relative hard disk drive number. This is the same number as hddn1 in the PDRIVE HDS parameter.
2. Reply the relative number of the first surface to be formatted. When formatting an entire hard disk drive, this value is 0.

3. Reply the number of recording surfaces to be formatted. When formatting an entire hard disk drive, the value is the number of recording surfaces the hard disk has (the RSC or TPC values discussed earlier). For Apparat 5, 10 and 15 Meg hard disks, this value is 2, 4 and 6 respectively. For Tandy 5 Meg drives, this value is 4.
4. Reply the number of tracks per surface UPS or tps1) for this drive. This is the same as the number of cylinders the drive has. For Apparat hard disks, this value is 306. For Tandy 5 Meg hard disk drives, this value is 153.
5. Reply the relative number of the first cylinder (the first track on a surface) to be formatted. When formatting an entire hard disk drive, this value is 0.
6. Reply the number of cylinders (number of tracks on each surface) to be formatted. When formatting an entire hard disk drive, this value is the same as given in #4 above.
7. Reply the track stepping rate code. Use a value of 15 here as we are not too concerned with a slow stepping rate during formatting.
8. Reply your intended data sectors per track. The normal value here is 32. The tracks supposedly have a capacity for 33 sectors per track, but test have shown that many parity errors occur. Specifying 32 sectors per track does allow for one error sector per track to be automatically specially encoded so that NEWDOS/80 will never see it.
9. Reply the sector interleave count. We recommend a value of 21 if there are to be 32 sectors per track. This value allows time for the DOS I/O routine, the transfer of the bytes on the cable to/from the drive's buffer, the actual read/write of the sector by the drive, and 1 to 2 milliseconds for the user program to invoke the I/O for the next sequential sector. This value of 21 is also optimal for the HDBACKUP program, which is too slow as it is. Values 19 and 20 will work, but allow much less time for the user program to turn the I/O around. Values 22 to 30 allow the user more turn around time but slowly decrease the number of I/Os per second that can be done. Values 0 - 18 allow too little time for the above functions and cause the hard disk to wait till the next revolution (16.7 ms) for the next sector.
10. Reply N if you wish to restart the specifications again at step 1 above. Reply Y if the program is to start the format.
11. Once started, the formatting will proceed, blinking an asterisk in the display upper right corner to indicate progress. If a track cannot be formatted with the required number of sectors, an error will be displayed giving the cylinder, head and number of error sectors above and beyond the number implicitly allowed in step 8 above. A track that has some error sectors and some good data sectors will have the good sectors numbered from track relative sector 0 consecutively on up with the higher numbered sectors for that track simply not there.

12. During HDFMTAPP execution, holding down the up-arrow key causes the program to terminate and the right-arrow key causes the program to pause. After right-arrow, pressing ENTER causes the program to continue. This pause/cancel function is useable only through the keyboard matrix, not via remote terminals.

13. When the format is complete, the number of tracks with too many errors will be displayed. If there are any such tracks, you SHOULD reformat the hard disk using a lesser sectors per track value. Mark the resulting sectors per track value spt1 on a label on the hard disk to remind you of what spt1 value MUST be used in all PDRIVE definitions for data volumes on that drive. HOWEVER, when only a small number of consecutive tracks have all the error sectors, you may decide to leave the error sectors alone and define your volumes (via PDRIVE) in such a way as to assure that the error tracks are not assigned to any volume (i.e., ending one volume on the last sector of the first good track preceding the bad track range and starting another volume on the first sector of the first good track following the bad track range). If the error tracks are assigned to a volume, NEWDOS/80 will give SECTOR NOT FOUND error when ever I/O is attempted to the a bad, non-existent sector. NEWDOS/80 does not maintain any bad track or bad sector tables.

14. If all tracks have been formatted with the required number of sectors, the hard disk is now ready for use by NEWDOS/80.

It is possible, due to the extensive specifications, for the HDFMTAPP program to format just one track on the hard disk. This may be of interest to a few users when a track has apparently gone bad and an attempt is to be made to reformat just that one track.

5. MOVING NEWDOS/80 VERSION 2.5 TO THE HARD DISK.

Usually, you want to have slots 0 to 3 as hard disk volumes and still have access to your two floppy disc drives. For this, it is necessary to operate using the NEWDOS/80 Version 2.5 system volume, which must be volume 0, from the hard disk. This section steps you through setting up NEWDOS/80 Version 2.5 to run from the hard disk. The hard disk is assumed previously formatted.

1. Be sure you know how to use the DOS command PDRIVE, especially with the Hard Disk Specification parameter HDS.

2. Mount a copy of the NEWDOS/80 Version 2.5 hard disk system diskette in floppy drive 0. This will be known as the system diskette as different from the hard disk system volume which will be on the hard disk.

3. Choose one of the system diskette's PDRIVE active slots whose number is greater than one. For this example slot 2 will be used (the SYSTEM option AL must be at least 3). If you choose a different slot number, then use that number in place of 2 in the following discussion.

4. Using PDRIVE,0,2,A,HDS=----- define floppy system diskette PDRIVE slot 2 with the specifications wanted for the hard disk system volume.

5. Execute the DOS command:

```
COPY, 0,2,,FMT,CBF,USD
```

and respond to the requests for SOURCE and DESTINATION diskettes (even though the destination is on a hard disk). GAT OVERFLOW error may occur if the spg1 value for the destination is less than that of the source; in which case you must increase the destination spg1 value.

6. Execute PDRIVE,2 to see the hard disk system volume's specifications for the 10 slots defined on that volume. Note that the definition for slot 2 has been duplicated in slot 0. This was done as a normal part of the COPY done above. Don't confuse the specifications of PDRIVE,2 which refers to system control data on drive 2, the intended hard disk system volume, with those of PDRIVE,0 which refers to system control data on drive 0, the floppy system diskette.

7. Using PDRIVE,2,----- define the PDRIVE specifications as you intend for that volume to be used as the system volume (drive 0). Since PDRIVE,2,2 has been duplicated as PDRIVE,2,0 in anticipation of that hard disk volume becoming the system volume, you MUST now redefine the PDRIVE,2,2 slot for another volume or by setting its vscl value to 0, causing slot 2 to be undefined. The specifications for PDRIVE,2 slots 0 - 3 must be for hard disk volumes only. Definitions for the floppies must be in slots 4 - 7 which correspond to the old drives 0 - 3 respectively. If one or more of the slots 4 - 7 are not used for floppies, then they may be used for hard disk volumes, thus allowing a maximum of 8 hard disk volumes to be active at any one time. Do not go on to the next step until all PDRIVE,2 slots have been defined as you will want them to be in the system operating from the hard disk, though it is not necessary to change any of them except slot 2 and you should not change slot 0. Remember, you cannot use PDRIVE parameter A when doing PDRIVE,2 definitions as that volume is not the current system volume.

8. Using SYSTEM,2,AL=xxx, specify the number of PDRIVE,2 slots to be active. xxx must be between 1 and 8, and must be at least 5 if any floppies are to be used.

9. The hard disk system volume now has the correct specifications, but we need a hard disk boot diskette (also known in this section as the boot diskette) to enable RESET (also known as BOOT), which must start on floppy drive 0, to switch to the hard disk system volume. This diskette must contain at least BOOT/SYS, DIR/SYS and SYS0/SYS, and must have its PDRIVE slot 0 defined exactly as for the hard disk system volume. So we proceed to do this.

10. If the system diskette's PDRIVE,0,1 specification is not identical to that for PDRIVE,0,0, then make them so by executing the command:

```
PDRIVE,0,1=0,A
```

11. Assign an otherwise unused diskette as the hard disk boot diskette and label it as such. Mount the boot diskette in floppy drive 1.

12. At this point, the system diskette is in floppy drive 0, the hard disk boot diskette in floppy drive 1, and the hard disk system volume is on the hard disk. Execute the DOS command:

```
FORMAT,1,,,,Y
```

13. When done, execute to DOS command:

`COPY,SYS0/SYS:2,:1`

to move a copy of SYS0/SYS, the resident DOS, from the hard disk system volume to the hard disk boot diskette. Since it is the first file placed on the boot diskette, aside from BOOT/SYS and DIR/SYS, it will automatically be placed in the proper place for RESET.

14. When done, execute: `PDRIVE,1,0=2` to move the proper hard disk system volume specification to the boot diskette's PDRIVE slot 0.

15. At this point, you may want to change the `PDRIVE,0,2` and `PDRIVE,0,1` definitions back to what they were before steps 3 and 10 above. This step is optional.

16. Remove the system diskette from drive 0. Move the hard disk boot diskette from drive 1 to drive 0 and press RESET. Computer execution will read the boot sector and then the resident DOS, SYS0/SYS, from the boot diskette in floppy drive 0 and then shift to the hard disk. You may now take the hard disk boot diskette out of drive 0 or leave it in, in which case it may be accessed via the PDRIVE slot 4 (used for floppy drive 0 when the hard disk system is in use) if `PDRIVE,0,4` is defined for a floppy. The diskette can be accessed by user programs as drive 4.

You may use the hard disk boot diskette as a normal data diskette by copying data files on to it. Remember though, it is the hard disk system's boot diskette and its SYS0/SYS is the resident DOS that is loaded into main memory at RESET time and remains there until the next RESET.

***** WARNING. A backup up of a hard disk boot diskette will not transfer its booting-up-the-hard-disk capability unless the backup is done using format 5 COPY with the BDU option.

The hard disk system volume is drive (slot) 0 when operating the system from the hard disk. The hard disk system volume does NOT have to be positioned at the beginning or a hard disk drive; in steps 4 and 5 above, you are allowed to place the hard disk system volume where you wish on the hard disk.

The file SYS0/SYS on the hard disk boot diskette MUST remain exactly identical to the SYS0/SYS on the hard disk system: volume. If you alter one, you MUST alter the other. This is necessary because the hard disk system: thinks its own SYS0/SYS is in the resident DOS area (4000H - 4CFFH and 0F900H - 0FFFFH) at all times when actually it is the SYS0/SYS from the hard disk boot diskette.

If you only have one floppy drive, then the following changes must be made to the above procedure:

1. Step 10 above is excluded.

2. In step 11, do not mount the boot diskette into drive 1.

3. In step 12, change the command to be `FORMAT,0,,,,Y` and perform diskette mounts as requested where the SYSTEM diskette is the system: diskette and the DESTINATION diskette is the boot diskette.
4. In step 13, change the command to be `COPY,$SYS0/SYS:2,:0` Perform the diskette mounts as requested where the SYSTEM diskette is the system diskette, SOURCE diskette is the hard disk system volume and the DESTINATION diskette is the boot diskette.
5. Replace step 14 with the following action. Enter SUPERZAP and at the menu, reply CDS. Remove the system diskette from floppy drive 0, and mount the boot diskette in floppy drive 0. Reply Y. Reply 2,2 for the source drive and relative sector. Reply 0,2 as the destination drive and relative sector. Reply 1 as the sector count. Press ENTER to return to menu. Remount the system diskette in floppy drive 0. Reply EXIT to exit SUPERZAP and return to DOS READY.

6. DEFINING PDRIVE SLOTS FROM A VOLUME DEFINITION FILE.

The definition of hard disk volumes via PDRIVE is more difficult and more critical than for floppy disk volumes. Therefore, it is recommended that the user carefully plan out his/her allocation of hard disk space amongst the various volumes and store the definitions (the HDS parameter part) into an ASCII text file (called a data volume definition file) created and updated by using either CHAINBLD or SCRIPSIT or both. Do this very, very, very carefully as you can create havoc amongst your data if two or more data volumes share the same hard disk sectors. Under NEWDOS/80 Version 2.5, you have great flexibility in assignment of hard disk space to data volumes, but with this flexibility comes complexity of definition.

Each record within the data volume definition file must start with a unique but arbitrarily assigned identification integer. Following the integer must be a comma followed by the intended PDRIVE definition excluding the initial part of the PDRIVE command (the PDRIVE,dn1,dn2, portion) and the ,A (for activation) as these parts of the PDRIVE command will be supplied by the EXTPDRIV/BAS program.

Since each definition record within the data volume definition file starts with an integer, you may imbed comments within the file as you like provided the comment record does not start with an integer.

It is strongly recommended that you keep copies of the data volume definition file on floppy diskettes in case that file on your hard disk becomes unusable. Remember, this is your master copy of the hard disk space layout!

Assuming that you have carefully constructed your data volume definition file, you may assign one or more of these definitions to the various PDRIVE slots when needed by running the BASIC program EXTPDRIV/BAS.

1. The program will ask for the filespec of your volume definition file and then open it.
2. The program will ask for the identification integer of the definition to be used. Respond with an EXACT copy of the integer that starts that definition's record in the file. The program will then search the file for the record.
3. When found, the program will ask for the two numbers needed for the PDRIVE,dn1,dn2,--- function. Respond with the two numbers separated by a comma. The first number, dn1, (usually 0) specifies which data volume contains the system control information, which will be changed by the PDRIVE command. The second number, dn2, specifies which PDRIVE slot definition is to be changed.
4. The program will then ask if slot definitions are to be activated within the resident DOS (i.e., the ,A PDRIVE parameter). Reply Y if so; N if not.
5. The program will then build the appropriate PDRIVE command and execute the command via DOS-CALL. You will see the PDRIVE results displayed.

6. The program will then ask if there is another definition from the same file to be applied. If you reply Y, the program returns to step 2 above. If you reply N, the program ends.

EXAMPLES of data volume definition file records:

1. 103,HDS=(0,153,0,4,32,6,0,2880,5,8,35,33)
2. 91,HDS=(1,153,0,4,32,6,1000,2000,0,8,0,33)
3. 44, TI=A, TD=E, TC=40, SPT=18, TSR=0, GPL=2, DDSL=17, DDGA=2

7. BACKING UP HARD DISKS TO DISKETTE:

Copies of user data stored on hard disk must be kept elsewhere in case the hard disk crashes, a program malfunctions or a user goof. Users MUST, from time to time, make backup copies of valued data, the frequency of backup depending upon how often the data changes and how valuable the data is.

NEWDOS/80 Version 2.5 provides the HDBACKUP (hard disk back up) function as a way of saving files from the hard disk(s) to floppy diskettes, and a way of restoring one, some or all of those files back onto the hard disk(s).

HDBACKUP saves by file rather than by full volume contents. It uses this considerably slower technique because over 50% of the restores that are eventually done involve only a selected set of files and not a full media or data volume. Restores to a hard disk don't have to be the result of a hard disk failure but more frequently are due to user mistakes or user program malfunction logically damaging or destroying certain files, and the restore should allow only the damaged files and their interrelated files to be restored, leaving unchanged all other files on the hard disk(s) involved. Unfortunately, saving by file requires more administrative consideration than does saving by entire volume contents; so we hope the greater flexibility will be worth it.

For purposes of HDBACKUP discussion, a backup is the content of the one or more diskettes used to contain the files copied from data volumes during the execution of the HDBACKUP program's SAVE function. These diskettes must be preformatted and, after being used by SAVE, cannot be read/written using standard DOS functions; however, they can be read/written using SUPERZAP disk (not file) mode.

In this discussion of the HDBACKUP function, a data volume refers to one of the active hard disk data volumes defined via PDRIVE.

HDBACKUP/CMD is the program that (1) creates a backup containing specified files from the various defined data volumes (as defined by PDRIVE) of your system, (2) lists which files are contained within a backup and (3) restores specified files from a backup to the various defined data volumes of your system. HDBACKUP is the method under NEWDOS/80 Version 2.5 of backing up your files from hard disk or diskette and, if necessary, restoring one, some or all of those files back to the hard disk or diskette. Under the SAVE parameter, HDBACKUP creates a backup that spans one or more diskettes. Under the LIST parameter, HDBACKUP lists the filespecs of and errors associated

with the files contained in the specified backup. Via the RESTORE parameter, HDBACKUP copies specified files from the backup to specified data volumes of your system.

The HDBACKUP SAVE function saves a file's contents, not its attributes. Except for the file name, name extension, data volume number and, if NND not specified, the logical record length, no other attributes of the file are saved such as passwords, protection level, etc. SYSTEM files are not SAVED. The user is responsible for backing 4p system files to regular diskettes using the COPT' command; normally it is sufficient to simply maintain, copies of your original NEWDOS/80 Version 2.5 Hard Disk System diskette and your regular NEWDOS/80 Version 2 System diskette. If the NND parameter is specified, system files included in the INCLUDE list are copied, but are no longer marked as system files.

Provided the NND parameter is specified, the HDBACKUP function is designed to attempt to run with TRSDOS-like DOSS other than NEWDOS/80 Version 2.5. Via the NED parameter, you must inform the HDBACKUP/CMD program of certain values for that DOS.

The HDBACKUP program requires passwords be disabled, as standard file OPENS are done without passwords in the filespecs. If passwords cannot be disabled in the current system, the passwords must be taken off the files being backed up. SYSTEM option AA=N disables passwords in NEWDOS/80.

The HDBACKUP program requires, unless the NND parameter is specified, that all volume directories be named DIR/SYS.

Usually after the user has responded to a request, HDBACKUP displays an * to indicate that it is no longer waiting for an operator response.

HDBACKUP blinks an * in the upper right corner of the display screen to let you know that is preceding in an orderly fashion. The speed of the blinks will vary due to the different functions.

The RESTORE function of HDBACKUP takes a very long time to initialize (in one test of 3444 files, it took 30 minutes). This extra initialization (1) performs KILLS if RENEW specified, (2) creates all new files, (3) CLOSES the files to store the new EOF and release any excess disk space on the data volume, (4) if NND not specified, writes the last sector of each file to allocate any needed disk space and (5) if NND not specified, updates the logical record length in the directory.

The HDBACKUP/CMD program expects the diskettes used for the backup to already be formatted. The program will write over the entire diskette; after SAVE, the diskette will not have a directory. The program will not tolerate a bad sector when writing to the backup diskettes. If a sector is bad, you have three options: (1) retry the write, (2) cancel the entire SAVE function, or (3) restart the SAVE function at the beginning of the current backup diskette. If you choose option 3, you will be asked for the current backup volume again; you should then (and not before) mount a different previously formatted diskette (remember to label it properly) and place the other diskette in your bad diskette collection.

The HDBACKUP command sequence is:

```
HDBACKUP
fc1
PRINT
NND=(filespec1,r/n,spg1,gpl1,spv1)
BSN=list1
TITLE=title1
DATE=datel
TIME=timel
SVL=list2
RVL=list3
SLOW
SKIP
RENEW
MAXERRS=ec1
TEST
INCLUDE
EXCLUDE
*END
```

HDBACKUP invokes the HDBACKUP/CMD program. HDBACKUP must be the only parameter on the first command line (the command line used by DOS to invoke the program). This program then displays the cursor and waits for the user to input subsequent command lines. Command parameters are processed until the *END parameter is encountered. There must not be extraneous characters within a command line. A command line may contain multiple parameters separated by commas, but a parameter must be fully contained within a command line. A command line is limited to 79 characters in NEWDOS/80 and 63 characters for most other DOSs.

The user will generally build the command lines and the file specifications for INCLUDE or EXCLUDE into a CHAIN (aka DO) file as it is strongly recommended that HDBACKUP commands be constructed very carefully. Though CHAINBLD will work, it is recommended you build your chain file via a word processor, storing the resulting file as an ASCII file.

**** Warning, be sure that the chain file has no extraneous characters after the end-of-line character for the SEND statement; otherwise subsequent responses needed for the HDBACKUP execution will receive bad data.

The TEST parameter was included to allow the user a 'dry' run to test the workability of the command parameters. If you don't know what your are doing, gain some familiarity by using the TEST parameter before doing a live run. Remember, you can't test a RESTORE until you have a backup to test with.

**** Warning, SAVE with TEST does write backup control information on the backup's 1st diskette; be sure that diskette is intended for a backup.

fc1 fc1 must be the first parameter after HDBACKUP. fc1 specifies the function to be performed which is one of the following:

1. SAVE Anew backup is created having title, date and time as specified by the TITLE, DATE and TIME parameters. The files specified, either explicitly or implicitly, are copied from the specified data volumes to as many backup diskettes as necessary. Parameters BSN, SVL and *END are required. Optional parameters are PRINT, NND, TITLE, DATE, TIME, SLOW, SKIP, MAXERRS, TEST, INCLUDE and EXCLUDE. If one of TITLE, DATE or TIME is not specified, the HDBACKUP program will ask for that parameter. If NND is specified, INCLUDE must be specified.

2. LIST This function lists the files contained within the specified backup and includes their associated error sector numbers. Required parameters are BSN and *END. Optional parameters allowed are PRINT, NND, TITLE, DATE and TIME. The listing starts with the backup's name, date, time, file count and error count. Then for each file in the backup's table of contents, the following are listed:

1. The filespec for the file.

2. If the file has been deleted from the backup table of contents, '***** DELETED *****' is displayed and steps 3 - 6 are bypassed.

3. The file's EOF value in xxx/yyy format where xxx is the relative sector within the file and yyy the relative byte within the sector.

4. The file's logical record length, 1 - 256.

If NND specified during the SAVE that made this backup, the record length may or may not be correct if the file's record length prior to the SAVE was not 256. This occurs under NND as HDBACKUP does not get the record length from the directory but records whatever record length appears in the FCE after OPEN. Normal NEWDOS/80 operations do not use the file's record length from the directory, but many users want it correct anyway. If a file's logical record length was changed during the SAVE and RESTORE, the user may correct it by using the LRL parameter of ATTRIB (see regular NEWDOS/80 Version 2 ZAP 007 (Model I) or ZAP 004 (Model III)).

5. The location within the backup of the file's header sector, expressed as a backup volume number and a relative sector within that volume. This is of interest only to those viewing/updating the backup via SUPERZAP. Volumes (diskettes) of a backup are numbered consecutively from 1, not 0.

6. If the file has any error sectors, they are listed each in the decimal format:

sssss/ee/vvv/rrrrr

where:

1. sssss is the sector's relative number within the file.

2. ee is the DOS error code.
3. vvv is the number of the backup volume containing the error sector
4. rrrrr is the sector's relative number within the backup volume.

3. RESTORE The files specified, either implicitly or explicitly, are copied from the backup to the specified data volumes. Required parameters are BSN, RVL and *END. Optional parameters allowed are PRINT, NND, TITLE, DATE, TIME, SLOW, SKIP, TEST, RENEW, INCLUDE and EXCLUDE.

PRINT This parameter informs the HDBACKUP program that display information is to be sent to the printer as well as the display. If PRINT is not specified, only the display will be used. If PRINT is specified, the program will display WAITING ON PRINTER, and then, if the printer is not ready, the program will hang.

NND=(filespec1,r/n,spg1,gpl1,spv1) This option specifies that the Disk Operating System (the DOS) is not NEWDOS/80 Version 2.5, though it can be. If NND is specified, the following hold:

1. SLOW is implied.
2. For SAVE, INCLUDE is required.
3. NND must be specified immediately after fcl and before BSN.
4. For RESTORE, the pre-allocation of needed file space during initialization is not done; an out-of-space error will not be detected until the file is actually restored.
5. file logical record lengths recorded in table of contents during SAVE or in the data volume directory during RESTORE may be wrong if they were not 256.

HDBACKUP is designed to run with NEWDOS/80 Version 2.5, but users initially may have their hard disk data under a different operating system, thus creating a dilemma, as NEWDOS/80 cannot process directories for other DOSs. Recognizing this as potentially a serious problem, an attempt (via the NND parameter) has been made to allow HDBACKUP to run under another DOS using faked extents in the FCB used for backup diskette I/O. This attempt will not work with a DOS that determines a diskette's characteristics from the diskette itself (as HDBACKUP writes over the entire backup diskette) or which automatically changes a drive's specification when an error is encountered. So far, the only successful tests have been (1) with Tandy's Model III Hard Disk Operating System (LDOS 5.1.3) using single sided, double density, 40 track drives as the backup drives specified in the BSN parameter with NND=(TEMPFILE:0,N,6,3,720), and (2) with Tandy's Model I Hard Disk Operating System (LDOS 5.1.3) using single sided, single density, 35 track drives as the backup drives specified in the BSN parameter with NND=(TEMPFILE:0,N,5,2,350). Apparat does not plan to test under the other DOSs or other configurations, and Apparat reserves the right to withdraw the NND parameter and all support for it at any time and without notice.

If using HDBACKUP with the NND parameter does not work with your other DOS, the user will have to find some other way of offloading the files from hard disk under the other DOS and reloading them under NEWDOS/80 Version 2.5.

***** Warning!!! Before using HDBACKUP to offload files under a DOS that is not NEWDOS/80 and then reloading the files to hard disk under NEWDOS/80, the user should offload the valued files to diskettes using the other DOS's normal backup procedures. This provides the user with a second backup source should the conversion to NEWDOS/80 fail.

When using the NND option, certain extra information MUST be provided to the HDBACKUP program. If you don't know what these values are, call the distributor for that DOS; don't call Apparat.

filespec1 is the filespec of new or existing file that HDBACKUP can write one sector to in order to determine a correct FCB to be used for backup diskette I/O. HDBACKUP will write garbage into that one sector and will not CLOSE the file. The file filespec1 must be for a file within a volume that is already mounted when HDBACKUP begins execution; further, for some DOSs, it may be necessary that the file be on a diskette with the same spg1, gpl1 and spv1 characteristics specified in this NND parameter. The diskette can be mounted on a drive specified in BSN below as HDBACKUP will conclude its use of file filespec1 before it asks for the first backup diskette.

r/n is one character, either R or N. R is specified if the EOF field of FCBs (the File Control Block in main memory, not the directory FDEs) for this DOS are in Relative Byte Address format (such as all NEWDOS versions and Model III TRSDOS 1.3). N is specified if the EOF field of the FCBs for this DOS are in Next Record Address format (such as LDOS (regular and hard disk), Model I TRSDOS 2.3 and Model III TRSDOS 1.1)

***** The choice of R or N is critical. Choosing the wrong value will cause every file not ending on a sector boundary to be assigned the wrong EOF in the backup, thus making the file one sector too long or too short. Further, reportable errors may occur.

Once again, the NEWDOS author apologizes for having brought Relative Byte Addressing to the TRS-80 world (the FCBs, not the directories) with the NEWDOS release in March, 1979, thus causing the confusion between RBAs and NRAs (Next Record Addressing). NRA was the standard at that time and has remained the LDOS standard (TRSDOS on the Model III changed to RBAs in July, 1982). NEWDOS shifted to and remains with RBAs because that method is the more reliable method for arbitrary random disk I/O.

spg1 is the number of sectors per granule for this DOS for the backup diskettes that will be mounted on floppy drive(s) specified in BSN below. (LDOS Hard Disk System uses spg1 = 5 for single density 5 inch diskettes and spg1 = 6 for double density).

gpl1 is the number of granules per lump for this DOS for the backup diskettes that will be mounted on the floppy drive(s) specified in BSN below. This is also known as granules per cylinder and is the

number of bits per byte used in the GAT sector to account for granule allocation. LDOS Hard Disk System uses gp11 = 2 for single sided single density 5 inch diskettes, gp11 = 4 for double sided single density, gp11 = 3 for single sided double density diskettes and 6 for double sided double density.

spv1 is the number of sectors per backup diskette. This is the total number of sectors on a diskette (720 for single sided, double density 40 track 5 inch diskettes, 350 for single sided, single density 35 track 5 inch diskettes, 1440 for double sided, double density 40 track 5 inch diskettes). Whatever the number, the DOS must be capable of doing I/O for that number of sectors per diskette.

HDBACKUP/CMD may be moved to another DOS via the following steps:

1. Under NEWDOS/80 Version 2.5, execute LMOFFSET. Respond D. Respond HDBACKUP/CMD. Respond new load address = 7000. Respond N to request appendage. Record the new start, end and entry address values displayed (will be used in step 4 below). Respond <ENTER> to indicate load point not being changed again. Respond N to keep DOS enabled. Respond D. Respond XXX/CMD:0 to write the modified module back to disk. Respond N. Respond PI again. You should now be back at DOS READY.

2. Execute the DOS command LOAD,XXX/CMD:0. This loads the load-offsetted HDBACKUP program created in step 1 into main memory from where it will be written to the other DOS's diskette in step 4 below.

3. Load the other DOS diskette into drive 0 and press RESET to bring up that DOS. Be sure that this DOS does not clear user memory upon coming up.

4. Use the DUMP command for that DOS to store onto that DOS's disk the HDBACKUP/CMD program loaded into main memory in step 2. The DUMP command will need the start, end and entry addresses recorded in step 1. See that DOS's manual for explanation of the DOS command DUMP. For LDOS, this command will be:

```
DUMP HDBACKUP/CMD:0 (START=X'start',END=X'end',TRA=X'entry')
```

where start, end and entry are the hexadecimal addresses recorded in step 1 above.

5. If that DOS's DUMP does not allow the filespec HDBACKUP/CMD:0, use what it will allow and then change the file's name via RENAME.

6. The HDBACKUP program is now ready for execution on that DOS.

BSN=list1 The Backup Slot Number specifies either one or two slot numbers (if two, list1 must be enclosed in parenthesis) of the slots (PDRIVE active volumes) to be used for reading/writing the backup diskettes.. These slots must be defined in PDRIVE as floppy disk drives. None of the backup slot numbers may be included in the volume numbers listed in the SVL or RVL parameters. If two slot numbers are specified, they must have the same PDRIVE definition. If only one slot is specified, all backup diskettes will be mounted as needed using that one drive. If two slot numbers are specified, the backup's volume 1 is left mounted on the first drive throughout the HDBACKUP function and the second drive is used for the other volumes. Since backup volume 1 is frequently referred to or updated during the SAVE or RESTORE, assigning two slots (drives) greatly reduces operator actions. If you only have two drives, run the system from the hard disk so that floppy drive 0 is free to be used as a backup drive.

TITLE=title1 title1 is the 0 to 48 printable character title of the backup. For SAVE, this title is assigned to the backup; if not specified in the command lines, the program will ask for it. For LIST and RESTORE, an error will be displayed if TITLE is not specified or title1 does not match that of the backup; the user may elect to use the backup anyway. Where TITLE is specified in a command line, it must be the last parameter of that line as title1, even if over 48 characters, is considered to be the rest of the line; the excess characters are ignored. During SAVE, when a backup diskette is first asked for, the program will reject the diskette if it has been used for a previous backup with the same title, date and time (as it may really be an earlier volume of this backup).

DATE=datel datel is the backup's date in nun/dd/yy format. For SAVE, this date is assigned to the backup; if DATE is not specified, the operator will be asked for it. For LIST and RESTORE, an error will be displayed if DATE is not specified or datel does not match the backup's date, but the user may elect to use the backup anyway.

TIME=timel timel is the backup's time in hh:mm:ss format. For SAVE, this time is assigned to the backup; if TIME is not specified, the operator will be asked for it. For LIST and RESTORE, an error will be displayed if TIME is not specified or timel does not match the backup's time, but the user may elect to use the backup anyway.

SVL=list2 This Save Volume List parameter is required for and used only if the function is SAVE. list2 specifies the volume(s) whose files are to be copied to the backup during SAVE. If list2 has more than one sub-parameter, list2 must be enclosed in parenthesis. list2 consists of one or more sub-parameters, separated by comas, of the type:

vn1 specifies the number of an active slot whose data volume files, as restricted by INCLUDE or EXCLUDE, are to be copied to the backup. vn1 may have integer values 0 to xxx, where xxx is one less than the SYSTEM AL parameter. vn1 must not equal a slot number specified in the BSN parameter.

RVL=list3 This Restore Volume List parameter is required for and used only if the function is RESTORE. This parameter specifies (1) volume numbers whose files in the backup, as restricted by INCLUDE or EXCLUDE, are to be restored and (2) optionally, the data volume to receive the files of another volume.

If list3 has more than one sub-parameter, list3 must be enclosed in parenthesis. list3 consists of one or more sub-parameters, separated by commas, of the type:

vr2=vn1 The files contained in the backup for volume vn1, as restricted by INCLUDE or EXCLUDE, are copied to volume vn2. Volume numbers in the INCLUDE or EXCLUDE list refer to vn1, not vn2. If vn2 and vn1 are the same volume number, the vn2=vn1 sub-parameter may be written as Just vn1. vn2 may have integer values 0 to xxx where xxx is one less than the SYSTEM; AL parameter. vn2 must not equal a slot number specified in the BSN parameter.

SLOW This option can only be used with NEWDOS/80 and specifies that the HDBACKUP program is NOT to use its faster diskette I/O when reading/writing the backup (not the data volumes) diskettes. SLOW is implied by NND. Normally, NEWDOS/g0 Version 2.5 uses a faster mode of backup diskette I/O in the hope of increasing the speed of SAVE and RESTORE by 20-40. SLOW should be specified only if the fast I/O appears to actually run slower than normal diskette I/O. You can study this by timing the time to read or write a backup diskette, preferably a volume other than backup volume 1.

SLIP During HDBACKUP processing when an error is encountered and the operator would normally have a 'SKIP' option allowing processing to continue, if the SKIP command parameter was specified, the SKIP option will automatically be assumed. Normally, this option will not be specified; however, there are times when a SAVE or RESTORE must accomplish what it can despite errors. For example, if part of a hard disk has gone bad and the disk is to be sent to the repair shop where it may or may not retain its current data, it may be important to assure that whatever data can be retrieved, is retrieved with the problem of restructuring bad files addressed later.

RENEW This option is used only with RESTORE. During HDBACKUP initialization after the files to be restored have been determined, a KILL is issued to the destination volume for each file that is to be restored. If the file did not exist on the destination volume, the KILL does nothing. Normal RESTORE initialization will then recreate the files on the destination volumes. The purpose of RENEW is to reallocate file space in, hopefully, less fragmented units (which can increase the efficiency of programs using these files); RENEW should only be used when all, or almost all, files of a data volume are being restored.

MAXERRS=ec1 ec1 is the number of errors the backup is to provide for in its error table. The default value is 640 with 6400 the maximum ec1 value allowed. MAXERRS is used only in the SAVE function.

TEST This option allows initialization processing to occur, including backup control information writes. When the initialization is done, HDBACKUP terminates with 'TEST COMPLETED' error. TEST allows the user to test the command parameters, including the INCLUDE or EXCLUDE lists.

*** Warning, TEST with SAVE writes control information to the backup's first diskette; this is necessary for a good test.

INCLUDE and EXCLUDE INCLUDE and EXCLUDE are mutually exclusive keywords. Each must terminate the current command input line. Subsequent command lines until but not including the *END command line compose a file list with each

line specifying either a volume number preceded by a colon (i.e., :3) or the filespec, without passwords, of a file to be INCLUDED or EXCLUDED. The number of volume numbers or filespecs allowed in a file list is limited by computer main memory constraints but is over 1500.

If the command line consist solely of a volume number, then all files for that volume are INCLUDED or EXCLUDED.

All volume numbers in the INCLUDE or EXCLUDE list must refer to a vnl volume number specified in the appropriate SVL or RVL parameter.

INCLUDE and EXCLUDE are optional keywords (except for SAVE with NND). If neither is specified, HDBACKUP will assume inclusion of all the files for the vnl volumes specified in the SVL or RVL parameter.

INCLUDE Only the files specified in the file list are included in the SAVE or RESTORE. If a file in the list does not exist, an error comment will be listed, and the operator given the option of bypassing the file.

EXCLUDE The files specified in the file list are excluded from the SAVE or RESTORE; all other files of the vnl volumes specified in the SVL or RVL parameter are copied. If a file in the list does not exist on the specified data volume (SAVE) or the backup (RESTORE), no indication is given to the operator.

*END This required parameter ends the HDBACKUP command specification.

INTERNAL STRUCTURE OF THE BACKUP:

For those users interested, this section will show the structure of a backup. Some users may find this description helpful in repairing a backup using SUPERZAP.

Each volume (diskette) of a backup has a volume header sector as the diskette's first sector. The header sector for volume 1 is the most important and is used by RESTORE and LIST to access backup control information. The headers for the other volumes contain roughly the same information, and are used during RESTORE to verify that you have mounted the correct volume and by SAVE to verify that you don't mount as a new volume for this SAVE a volume that has already been used in the SAVE. The user must remember that a file's sectors can span many backup volumes and must allow for the volume header records when computing where a particular sector of a particular file is within the backup. The contents of the backup volume header sector are:

1. 48 byte backup title.
2. 8 byte backup data in mm/dd/yy format.
3. 8 byte backup time in hh:mm:ss format.
4. 2 byte count of sectors for the table of contents.
5. 1 byte count of sectors for the error table.
6. 2 byte count of files in the table of contents.
7. 2 byte count of number errors allowed during SAVE.
8. 2 byte count of sectors per backup volume.

9. 2 byte value = this diskette's volume number.
- ***** valid only for volume 1:
 10. 1 byte of control bits:
 - bit 7 = 1, the SAVE is complete.
 - bits 6 - 0, undefined and reserved, must be 0.
 11. 2 byte count of errors in error table.
 12. 2 byte count of volumes for this backup.
 13. 3 byte backup total sector count.
14. remainder of sector's bytes are 00H.

On backup volume 1, the table of contents sectors immediately follow the volume header sector. Each sector contains eight 32 byte file entries of the form:

1. 3 byte file name, padded on right with blanks.
2. 3 byte file Filename extension, padded on right with blanks.
3. 1 byte data volume number.
4. 3 byte file EOF in FBA format.
5. 1 byte logical record length (0=256). Not necessarily valid if NND specified during SAVE.
6. 3 byte relative sector number within the backup of the file's header sector.
7. 2 byte relative entry number of this entry within the table of contents.
8. 1 byte control bits:
 - bit 7 = 1, this table of contents entry is used.
 - bit 6 = 1, this file is active.
 - Bit 7-6 = 10, file has been deleted from the backup. Actually some of it may still be there, but LIST and RESTORE ignore it.
 - bits 5 - 0, undefined and reserved, must be 0.
9. The remainder of the 32 byte entry are bytes 00H.

The error table sectors immediate follow the table of contents. Each sector has 64 entries of the form:

1. 1 byte containing the DOS error code plus 40H. If the byte is 00H, the error has either been corrected by the user or he/she wants it ignored.
2. 3 byte relative sector number within the backup of the file sector in error. If the error is corrected or to be ignored, this value must be set to 0.

The remainder of the backup is file data with each file's sectors preceded by a file header record. If a file's EOF is zero, then only the file's header record will appear. The user must remember that where a file's sectors flow onto the next backup volume, the first sector on that volume will be the volume's header sector, not a file sector. The format of a file header is:

1. The first 22 bytes are an exact copy of the first 22 bytes of the table of contents entry for this file, but with none of the changes to the entry after it was initially created. During RESTORE, these 22 bytes of the file header must match the 22 bytes from the table of contents.

2. Each of the remaining bytes of the file header sector contains the ones complement of its relative location in the sector. This makes it easier to recognize a file header should it be necessary to search for it.

HDBACKUP EXAMPLES:

1. HDBACKUP
SAVE,BSN=(4,5),SVL=(0,1,2,3),*END

This is a copy of user files from hard disk data volumes 0, 1, 2 and 3 to a backup whose diskettes will be mounted on the floppy drives associated with slots 4 and 5 (assumed defined for floppy drives 0 and 1 respectively), with backup volume 1 remaining on slot 4's drive and the other backup volumes requested on slot 5's drive as needed. The user must have on hand enough pre-formatted diskettes for the needs of the backup. Since slots 4 and 5 are the access to floppy drives 0 and 1, we know that the hard disk system is being run from the hard disk.

2. HDBACKUP
RESTORE,BSN=(4,5),RVL=(0,1,2,6=3),*END

This is a copy of user files from a backup to data volumes 0, 1, 2 and 6. The backup diskette volumes will be mounted on the drives for slots 4 and 5 as described in the above example. All files in the backup are copied, but the files that originally came from volume 3 are actually written to volume 6 instead.

3. HDBACKUP
SAVE,BSN=1,SVL=(2,3,4,5,6,7)
EXCLUDE
XXX/DAT:4 YYY/DAT:6,*END

This is a copy of all user files from hard disk data volumes 2, 3, 4, 5, 6 and 7 to backup diskettes which will all be mounted as needed on the floppy drive 1. File XXX/DAT of volume 4 and file YYY/DAT of volume 6 will not be copied to the backup. Since BSN=1 was used for the backup floppy drive, we know the system is being run from a system diskette in floppy drive 0.

4. HDBACKUP
LIST,BSN=1,PRINT,*END

The contents of the backup's table of contents is listed on both the display and the printer.

5. HDBACKUP
SAVE,BSN=(4,5),SVL=(1,2),INCLUDE
ACCTPYBL/DAT:1
ACCTRVBL/DAT:1
PAYROLL/DAT:2
INVENTORY/DAT:2
*END

A backup is made consisting only of the 4 files specified in the INCLUDE list. In this installation, the burden of making backups of valued files has been placed on the individual users, in this case, accounting.

```

6.      HDBACKUP
        SAVE,NND=(TEMPFILE:0,N,6,3,720),BSN=(4,5)
        SVL=(0,1,2,3),INCLUDE
        FILE001:0
        FILE002:0
        FILE003:1
        and so on through
        FILE999:4
        *END

```

In this example, the HDBACKUP/CMD program has been previously moved to the LDOS Hard Disk Operating System (in the manner described at the NND discussion). The HDBACKUP runs under LDOS 5.1.3 and dumps the specified files from volumes 0, 1, 2 and 3 to 4 backup whose diskettes have all been preformatted as single sided, double density, 5 inch 40 track (with spg = 6, gpl = 3 and spv = 720). After the hard disk has been reinitialized for NEWDOS/80 Version 2.5, the HDBACKUP program under NEWDOS/80 (without the NED parameter) can be used to RESTORE the files from the backup to the hard disk.

When NND is specified for a SAVE, such as above, an INCLUDE list must be used to inform the HDBACKUP program of which files to copy to the backup, as the HDBACKUP program does not read the non-NEWDOS/80 directories.

This example could be used for single sided, single density 35 track backup diskettes under LDOS 5.1.3 on the TRS-80 Model I by using replacing the NED parameter with NND=(TEMPFILE:0,N,5,2,350). If 40 track diskettes are used, replace 350 with 400.

The TEMPFILE:0 filespec used in this example is just our choice of a filespec for this example; you are free to use any filespec you wish so long as it conforms to the specifications given for the END parameter.

Index

- A -		POPR	7-12
		POPS	7-12
		SASZ	7-12
ACC	2-4	SS	7-14, 12-9
alpha	10-1	SWAP	7-13
alphanumeric	10-1	I	7-10
APPEND	2-2	J	7-10
ASC	2-4, 2-19	L	7-10
ASE	2-4, 2-19	O	7-10, 7-14
ASPOOL	5-3, 6-19	P	7-10
activation	6-21	R	7-10
initial setup	6-19	S	7-10
Asynchronous Execution	2-4	T	7-10
ATTRIB	2-3	X	7-10
AUTO	2-5	Z	7-10
- B -		doscmd	7-11
		COPY	2-9, 12-4, 12-9
BASIC MODULES	5-2	CREATE	2-18
BASIC2	2-5	CVD	8-20
BAUD	2-44	CVI	8-20
BDU	2-13	CVS	8-20
bit	10-1	- D -	
BLINK	2-5	DATE	2-19, 3-11
BOOT	2-6, 10-1	DDGA	2-15
BOOT/SYS	5-1, 10-1	DDND	2-12
BREAK	2-6, 12-2	DDSL	2-15
buffer	10-1	DEBUG - 123	2-20, 4-1, 3-3, 12-2
byte	10-1	DEC	10-2
- C -		DFG - MINI-DOS	4-6
		DFO	11-8
CBF	2-14	DI	7-4
CHAIN	2-6, 4-7	DIR	2-20
CHAINBLD	5-3, 6-16	DIRCHECK	5-3, 6-12
chaining	10-1	directory	12-2, 10-2
CHAINTST	5-3	Directory Structure	5-4
character	10-1	DIR/SYS	5-1, 10-2
CHNON	2-7	DISASSEM	5-3, 6-5
CFWO	2-14	DISK BASIC	7-1, 8-1
CLEAR	2-8	activating	7-2
CLOAD	7-1	command truncation	7-4
CLOCK	2-9, 3-11	direct commands	7-3
CLOSE	3-7, 10-2, A-9	enhancements	7-1
CLS	2-9	I/O enhancements	8-1
CMD	7-8	file types	8-1
A	7-8	module overlays	7-1
B	7-8	DO	2-22, 4-7
BREAK	7-1	DOS	10-2
C	7-8	DOS-CALL	4-12, 3-4, 10-2
D	7-9	DOS command (doscmd)	10-2
E	7-9	DOS ROUTINES	3-1
F	7-9	DOS SYSTEM MODULES	5-1
DELETE	7-13	DPDN	2-10
ERASE	7-12	DU	7-4
KEEP	7-12	DUMP	2-22
POPN	7-12		

- E -

EDTASM	5-3,6-14
EDIT direct commands	7-1,7-3
/ or shift up-arrow	7-3
; or shift down-arrow	7-3
.	7-3
,	7-3
:	7-3
@	7-3
up-arrow	7-3
down-arrow	7-3
EOF	10-3
EOL	10-3
EOM	10-3
EOR	10-3
EOS	10-3
ERROR	2-24,3-2
error messages	9-1,7-1
DOS	9-1,7-1
BASIC	9-2,7-2
extent element	10-3

- F -

fan	10-3
FCB	5-9,3-9,3-10,10-3
FDE	5-6,10-3
FF FILE	8-10,10-3,A-39,B-5,B-6,B-7
FI FILE	8-10,10-4,A-45,B-15
FIELD ITEM FILE	10-4
file	10-4
file item	10-4
filearea	10-4
filespec	10-4
FILE TYPE (ft)	8-10
FI	8-10,A-45
FF	8-10,A-39
MI	8-10,A-35
MF	8-10,A-30
MU	8-10,A-20
FILE POSITIONING (fp)	8-3,10-5,A-1
FIXED ITEM FILE	8-7,10-4
FMT	2-12
FORMAT	2-24,12-9,10-4
FORMS	2-26
FPDE	5-7,10-5
FREE	2-27
FXDE	5-9,10-5

- G -

GAT sector	5-5,12-2,10-5
GET	8-12,A-10
granule	10-5

- H -

hash code	10-5
hexadecimal	10-5

HIMEM	2-27,12-8,10-6
HIT sector	5-6,10-6

- I -

I/O error recovery	8-19
I/O link or path	10-6
ILF	2-14
IGEL	8-4,10-6
IGEL expression	8-5,10-6
IGELSN	10-6
item group	10-7

- J -

JKL	2-27,4-13
-----	-----------

- K -

KDD	2-13
KDN	2-13
KILL	2-28

- L -

LC	2-29
LCDVR	2-29
len	10-7
LIB	2-30
LINES	2-26
LIST	2-30
LMOFFSET	5-3,6-9
LOAD	2-31,3-7,7-4
V option	7-4
LOC	8-18,A-18
LOCK	2-3,2-40
LOF	A-17
logical record	10-7
Lower Case Suppression	7-8
LRECL	10-7
LRL	2-18
LSET	8-20
LUMP	12-2,10-7

- M -

MARKED ITEM FILE	8-7,10-7
MDBORT	2-31
MDCOPY	2-32
MDRET	2-32
MERGE	7-5
MF FILE	8-10,10-7,A-30,B-12,B-14
MI FILE	8-10,10-7,A-35, B-14,B-15,B-17

MINI-DOS - DFG	4-5
MKD\$	8-20
MKI\$	8-20
MKS\$	8-20
ms	10-7
MU FILE	8-10,10-7,A-20 ,B-2, B-3,B-4,B-9,B-10,B-11

- N -		REF	2-40
null	10-7		
null character	10-8		
null string	10-8		
NDNW	2-12	sector	10-9
NDN	2-13	SETCOM	2-44
NDPW	2-12	SN	2-13
NFMT	2-12	SOR	10-9
NOWAIT	2-44	SPDN	2-10
		SPW	2-12
		STMT	2-45
		SUPERZAP	5-3,6-1
		display mode	6-3
		function mode	6-1
		modify mode	6-4
		SCOPY	6-3
		SYSTEM	2-45,12-3
		AA	2-46
		AB	2-46
		AC	2-46
		AD	2-46
		AE	2-46
		AF	2-46
		AG	2-46
		AH	2-46
		AI	2-47
		AJ	2-47
		AK	2-47
		AL	2-47
		AM	2-47
		AN	2-47
		AO	2-47
		AP	2-47
		AQ	2-47
		AR	2-47
		AS	2-48
		AT	2-48
		AU	2-48
		AV	2-48
		AW	2-48
		AX	2-48
		AY	2-48
		AZ	2-48
		BA	2-48
		BB	2-48
		BC	2-49
		BD	2-49
		BE	2-49
		BF	2-49
		BG	2-49
		BH	2-49
		BI	2-49
		BJ	2-49
		BK	2-49
		BM	2-49
		BN	2-49
		SYSTEM Files Required	5-1
		SYSTEM reduced size	5-4
- O -			
ODN	2-1 2		
ODPW	2-14		
OPEN	8-9,3-5,3-6,9,10-8,A-6		
- P -			
PARITY	2-44		
partial record I/O	10-8		
PAUSE	2-33		
PDRIVE	2-33,12-2		
A	2-37		
DDGA	2-37		
DDSL	2-37		
GPL	2-37		
SPT	2-37		
TC	2-36		
TD	2-36		
TI	2-34		
TSR	2-37		
PFST	2-25		
PFTC	2-25		
PRINT	2-39		
print/input file	10-8		
PROT	2-3,2-40		
PSEUDO FIELD	8-17		
PURGE	2-41		
PUT	8-14,A-13		
- R -			
R	2-41		
RBA	12-1,10-8		
REC	2-18		
REF	7-7		
REGISTRATION	1-1		
REMB A	8-16,10-8		
REMRA	8-16,10-8		
RENAME	2-42		
RENEW	7-17		
RENUM	7-5		
Reporting errors	11-1,11-2		
reset/power-on	10-8		
ROUTE	2-42,12-8		
RSET	8-20		
RUN	7-4		
V option	7-4		
RUN-ONLY	7-2,7-8		

STOP 2-44

- T -

track 10-9
TIME 2-50
timer interrupts 3-3,3-4

- U -

UBB 2-13
UDF 2-4
UNLOCK 2-40
UPD 2-4,2-14
UPDATE SERVICE 11-6
USD 2-13
USR 2-14,2-41
user segmented file 10-9

- V -

VERIFY 2-51
vice 2-44

- W -

WIDTH 2-26
whole record I/O 10-9
WORD 2-44
WRDIRP 2-52

- X -

XLF 2-14

- Z -

ZAP 10-9
ZAPS
 Distribution 11-5
 Duplication 11-7
 Format 11-2
 Installation 1-4,11-5,11-6
 Procedure 11-4
 Update Service 11-6

- SYMBOLS -

/ext 2-14,2-41
*name routine 3-10,3-11
123 - DEBUG 2-19,4-1
/ or shift up-arrow 7-3
; or shift down-arrow 7-3
. 7-3
, 7-3
@ 7-3
up-arrow 7-3
down-arrow 7-3