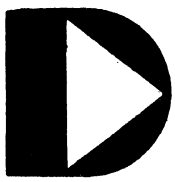
DISK OPERATING SYSTEM
DOS.
User's Guide

Version 2

January, 1976

Model Code No. 50216

DATAPOINT CORPORATION



The Leader in
Dispersed Data Processing

PREFACE

The purpose of this User's Guide is to provide the user of a Datapoint DOS that information required to generate a system, make effective use of the available commands, and to make user-written programs compatible with the DOS.

This manual applies to all Version 2 "dot-series" Disk Operating Systems, such as DOS.A, DOS.B, etc. Additional information concerning unique characteristics of a particular DOS is provided in the appropriate Version 2 DOS System Guide.

TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION

Datapoint Corporation's Disk Operating System (usually abbreviated DOS) is a comprehensive system of facilities for sophisticated data management.

DOS provides the operator with a powerful set of system commands by which the operator can control data movement and processing from the system console. These commands allow the system operator to accomplish in a very short time things which would be substantially more difficult on much larger computing systems. Sorting a large file, for instance, can generally be accomplished in one single command line: compare this with the bewildering pile of system commands required to perform a similar function on other computers! In spite of the simplicity of operation, even the most sophisticated personnel will be surprised at the wide range and versatility of features provided.

To the programmer, DOS offers a large set of facilities to simplify and generalize his task and file management problems. Such advanced concepts as completely dynamic disk space allocation allow programs to efficiently operate without regard to the amount of space required for the data files they are using. In addition, the very efficient disk file structure used by DOS allows for direct random access to data files at speeds comparing very favorably with even the largest mainframes. The standard use of fully space-compressed text files allows source programs and many data files to fit in half or less of the disk space that would normally be required on larger systems.

For the systems analyst and systems designer, DOS provides the solid foundation for powerful and sophisticated packages such as Datapoint Corporation's highly successful DATASHARE and DATACCOUNTANT systems.

Programmers and operators alike will appreciate the automatic program chaining facility provided by the CHAIN command of the DOS. Programmers will enjoy using CHAIN because it enables the creation of complete, sophisticated job files which allow the automatic execution of an almost unlimited number of job steps, all without operator intervention at the keyboard. Ease of assembling or compiling a large system of programs is just one of the many benefits achieved by use of the CHAIN facility. Operators will appreciate CHAIN because entire data processing tasks can be queued for execution and invoked with only a single

command line to the system.

These features, combined with the ability to support up to 200 million bytes of high-speed random access disk storage, provide a full range of data processing capabilities unmatched by any comparable business-oriented system.

1.1 Hardware Support Required

The minimal configuration required to run DOS is a Datapoint 1100, 2200, 5500 computer, with a minimum 16K of memory, and one (9350, 9370, or 9380 series) disk storage unit. For backup and support purposes, users with the Diskette 1100 computer are required to have at least one system with more than one diskette drive. Configurations based on the other processors can operate with only a single disk drive unit in conjunction with the integral tape cassettes, but for backup and system support purposes a two-drive system is a strongly recommended minimum.

The two 5500- only DOS, DOS.D and DOS.E, support a minimum of two physical disk drives.

Users running single physical drive 9350, 9370, and 9380 configurations are supported under DOS.A, DOS.B, and DOS.C respectively.

1.2 Software Configurations Available

DOS is provided in several different versions. Different versions are used depending upon the type of disk in use at an installation. Specific versions are indicated by a letter after a period in the name of DOS. As an example, the following versions of DOS are currently defined:

DOS.A -- Supports 9350 series disk drives on Datapoint 2200 and 5500 series computers.

DOS.B -- Supports 9370 series disk drives on Datapoint 2200 and 5500 series computers.

DOS.C -- Supports 9380 series disk drives on Datapoint 1100, 2200 and 5500 series computers.

DOS.D -- Supports two or more 9370 series disk drives (with 16 buffer disk controller) on 48K Datapoint 5500 series computers.

DOS.E -- Supports two or more 9350 series disk drives (with 16 buffer disk controller) on 48K Datapoint 5500 series computers.

1.3 Program Compatibility

This manual describes the compatible set of facilities available to the DOS user within the Disk Operating System. Programs written in any of the supported higher level languages (Datashare, RPG II, BASIC, etc.) will generally run unmodified on any of the DOS. Most programs written in assembler language will also run under any of the dot-series DOS, without reassembly.

Basically, in only a few isolated cases will a program need to be changed when it is transferred from one DOS to another. The need for program modification will usually stem from one or more of the following types of situations, which should obviously be avoided whenever possible:

1) Programs which make assumptions regarding the size of files. For example, programs originally written for the 9350 series disks might assume that the size of the biggest possible file could be expressed as four ASCII digits. Under DOS.B, this assumption is invalid since files under DOS.B may be over 30,000 data sectors long.

2) Programs which make assumptions regarding the physical structuring of the data on the disks. For example, each DOS allocates space on the disk in segments of different sizes, and places its system tables in different locations on the disk.

3) Programs which generate or modify physical disk addresses themselves. Since the disks are each organized somewhat differently to take advantage of the particular characteristics of the specific type of drives involved, the physical disk address formats naturally vary among different DOS.

4) Programs which rely upon other characteristics of a DOS which are not documented in this manual. A possible situation would be where a programmer might look at the values in the registers following the return from a system routine and determine, for instance, that some routine always seemed to return with the value "1" in one of the registers. If he then constructs his program in such a manner that it will not function correctly if the "1" is not present upon return from the routine, then he is obviously likely to find that his program will not work properly on a different DOS.

All of the above situations, except for the first, will usually only occur in assembler language programs operating at the

very lowest levels. Programmers who for their application require this level of detailed knowledge about the DOS will find the information specific to each DOS in the DOS System Guide corresponding to the DOS they are using.

The DOS System Guide for a specific DOS is also the place where one will normally turn for operational details and information about the hardware and software specific to his particular configuration. For example, the commmand INIT9370 (to format a disk volume for use in the 9370 series disk drives) is described in the DOS.B and DOS.D System Guides, since it is clearly not applicable to users of the 9380 series flexible diskette drives.

CHAPTER 2. OPERATOR COMMANDS

All Datapoint computers include, as a standard feature, an integral CRT display unit through which the internal computer communicates with the operator. The system console also includes a typewriter-style keyboard which the operator employs to communicate with the computer. The DOS is normally controlled by commands entered at this system console.

When DOS first "comes up", (computer jargon for "become ready for commands",) it displays a signon message on the CRT and says "READY". At this point DOS is ready to accept a command line. This command line, typed by the operator, tells DOS which program one wants to run and may name one or more files on disk which the program is to use. These files could be program files (files containing programs in one form or another) or data files (files containing data to be used by executing programs). If, as an example, the user wished to edit a program file on his disk, he would simply type:

```
EDIT PROGNAME
```

where "PROGNAME" is the name of his program. EDIT is a DOS command which allows the user to edit files stored on the disk.

A large assortment of useful commands is provided with DOS. These include the DOS editor and many useful disk file handling commands. A complete set of CTOS compatible cassette handling commands are also provided, allowing the transfer of files between the disk and cassettes.

Since the commands are actually programs which the system loads and executes to perform the task required, the command language is naturally extensible to include any program desired, thus leading to a powerful keyboard facility.

CHAPTER 3. ABOUT DISK FILES

Each of the DOS-supported disks stores information in the form of sectors, each of which contains 256 bytes of information. Each byte is capable of storing one ASCII (or EBCDIC) coded character. Information stored in these sectors is usually grouped with a number of other sectors containing related information, and together this group is referred to as a file.

3.1 File Names

Files are identified from the console by a NAME, EXTENSION, and LOGICAL DRIVE NUMBER. The NAME must start with a letter and may be followed by up to seven alphanumeric characters. Examples of typical file names are:

EDIT
PAYROLL
EMPLOYEE
JUL1075
MONDAY
LEDGER
etc.

The EXTENSION must start with a letter and may be followed by up to two alphanumeric characters. It further defines the file, usually indicating the type of information contained therein. For example, TXT usually implies data or source files (e.g. DATASHARE, ASM or SCRIBE source lines), ABS usually implies program object code records that can be loaded by the system loader, and CMD usually implies programs that implement commands given to DOS from the keyboard. Most commands have default assumptions concerning the extensions of the file names supplied to them as parameters. However, extensions may otherwise be considered as an additional part of the name.

The LOGICAL DRIVE NUMBER specifies which logical drive is to be used. It is given in the form :DR(n) or :D(n), where (n) is zero through the maximum number of drives supported by the specific DOS in use. If the drive is not specified, DOS generally searches all drives starting with zero. Note that each logical drive contains its own directory structure. Specifying the drive number enables one to keep programs of the same NAME and EXTENSION on more than one drive. In addition, specifying a logical drive

allows creating files on any logical drive desired.

3.2 File Creation

Files are always created implicitly. That is, the operator never specifically instructs the system to create a given file. Some commands create files from the names given as their parameters. Since space allocation is dynamic, the operator never specifies how many records a file will contain, or where on a disk the file is to be located.

3.3 File Deletion

Deleting files is made somewhat more difficult to prevent the accidental destruction of valuable data. Files can be protected against deletion or both deletion and modification. In addition to this, the operator must always explicitly name the file he is deleting and even then must answer a verification check stop before the actual deletion occurs.

For example, if an operator wished to delete a file called OLDPROG/ABS, he would enter the following dialogue (where operator commands and replies are indicated in lower case, although they would be entered in upper case):

```
kill oldprog/abs
THAT FILE IS OLDPROG/ABS (103) ON DRIVE 2
ARE YOU SURE? yes
* FILE DELETED *
READY
```

3.4 Program Execution and File Specifications

DOS has no explicit RUN command. To execute a program, its name is entered as the first file specification on the command line. This is the mechanism by which system commands and other programs alike are executed. The first file specification may be followed by several more, depending upon the requirements for parameterization of the program being run. A file specification is of the form:

```
NAME/EXTENSION:DRIVE
```

where any of the three items may be null (except that the NAME must be given in the first specification, which denotes the program to be run). Note that the "/" indicates that an extension follows and the ":" indicates that a drive specification follows. If either of these items is not given, the corresponding

delimiting character is not used. For example:

```
NAME/ABS:DR0
NAME/ABS
NAME:DR0
NAME
```

are all syntactically correct. File specifications may be delimited by any non-alphanumeric that would not be confused with the extension and device indicators ("/" or ":") or the option delimiter (";"). For example:

```
COPY NAME/TXT,NAME/ABS
COPY NAME/TXT NAME/ABS
COPY NAME/TXT/NAME/ABS
```

will all perform the same function. If an extension is not supplied in the first file specification, it is assumed to be CMD. In the above examples, COPY/CMD is used for the complete file name sought in the directory for the command program. If one wanted to run a file he had created with extension ABS, he would simply enter:

```
NAME/ABS
```

and his program would be loaded and executed. If the name given cannot be found in the directory or directories specified, the message:

```
WHAT?
```

will be displayed. DOS can load any object code at or above location 01400 (octal).

CHAPTER 4. SYSTEM GENERATION

Upon initial installation of a new Datapoint system, the user will generally start off with several brand new disks. Before a disk can be used by the DOS, however, the disks must be prepared; this process is known as DOS generation or DOSGEN.

Some types of disks require special treatment even before a DOSGEN can be properly done on them. One example is disks that are used with the Datapoint 9370-series disk drives. On these disks, formatting information must first be written onto the disk. Such special treatment, to be done before the DOSGEN process, is described in the DOS System Manual for the specific DOS in use; only the general-case DOSGEN process will be described here.

There are two methods for doing a DOSGEN on a disk. They differ primarily by whether the DOS is generated from the very beginning (e.g. from a cassette tape) or from an up and running DOS system. People doing their very first DOSGEN or having only one physical disk drive will have to DOSGEN using the cassette DOSGEN approach; otherwise they will be able to use the generally faster disk DOSGEN command supplied with the DOS and described later.

4.1 DOSGEN from cassette

Cassette DOSGEN requires a DOS generation cassette package, which contains the DOS to be generated, and a disk onto which to generate the DOS.

Each physical disk drive has a number associated with it, which is called the physical drive number. At the time the drive is installed, the Datapoint Customer Service engineer will demonstrate the proper technique for inserting and removing disks from the disk drives and will indicate which numbers are associated with which drives.

The two cassette tape decks on top of Datapoint computers (that are so equipped) are usually referred to as the front deck and the rear deck. The rear deck is the one physically closer to the row of cooling slots on the top of the computer and toward the back. In disk oriented systems, this rear deck is almost invariably used to hold a tape known as the DOS boot tape. The front deck is the deck physically closer to the keyboard. In disk oriented systems, this front deck is almost invariably used to

hold cassettes which either contain data to be processed with one or more of the available system commands, or blank cassettes to which data can be written via appropriate system commands.

Another important hardware feature is the read-only switch present on the 9350 and 9370 series disk drives. (There is such a switch on the 9380 series drives also, but on these the switch is internal to the controller and for maintenance use only). This switch is usually labelled with something descriptive such as "Read-write/Read only" or "Protect". These switches physically prevent the disk controller from writing on the disk. Since the DOSGEN process obviously needs to write on the disk (as do most DOS operations) it is important that these switches be set to allow writing. See the appropriate DOS Systems Guide for a discussion of the use of the Write-Protect switch.

After a disk is in place and spinning and the DOSGEN cassette is in place in the rear deck, load the DOSGEN program from the rear deck by pressing the key on the computer keyboard marked RESTART. (Datapoint 5500 users must also press RUN at the same time for the RESTART key to have effect). If the tape stops moving and the STOP key light comes on then the tape probably did not load correctly. Usually this will occur within about 5-10 seconds after the tape is rewound and starts moving forward. If this halt occurs, the procedure should be repeated as necessary. If after several tries the DOSGEN program still does not load, the tape may be bad and should be replaced.

When the program has loaded, it will display a signon message. It will then try to make sure a disk containing valuable information will not be accidentally overwritten. The program then asks if the user wishes a full generation or just the replacement of the system and utility files. The full generation does a quick check of the entire disk by writing to and/or reading from it. As each question is asked, the operator is required to key in his answer (usually "Y" or "N" is sufficient) and to terminate his response with the ENTER key (by DOS convention almost all entries to questions posed by programs are terminated by ENTER).

After the cylinders used by the system files have been checked the program will ask if the user wants to lock out any cylinders. If the user wants to set aside an area of the disk for abnormal use, (or wishes to prevent the use of a portion of the disk which may be bad) then he should reply "Y" to this message. In this case the operator is asked which cylinders he wishes to lock out; the reply should be of the format:

12,14,20-28

The above example would cause cylinders 12, 14, 25, 26, 27, and 28 to be locked out. Note that the cylinder numbers to be locked out are given in decimal as opposed to octal. However,the normal answer to this question will be "N".

Following this, the disk is checked for obvious bad spots and these places are then automatically locked out. When the surface checking has finished, the DOS and a few commands are copied to the disk. When enough of the DOS has been copied to the disk to bring the system up, the DOS is brought up and the standard DOS signon message and "READY" are displayed.

Important: The DOSGEN procedure is not completed until the commands have been loaded onto the disk.

Before loading the commands, first place a blank cassette in the front deck and type "BOOTMAKE" at the console. Follow the instructions that are subsequently displayed and a DOS "Boot block" will be written onto the front tape. It is probably a good idea to repeat this process several times to ensure getting a good boot tape before proceeding. These boot tapes are the mechanism by which the DOS is "brought up".

The next step is to load the commands. Place the first of the commands tapes into the front deck. The message "READY" should at this point appear on the display. If it does not, take one of the boot tapes generated in the previous step, place it into the rear deck and load it just like the DOSGEN tape. After several seconds the DOS signon and "READY" should appear. If they do not and the "Stop" light comes on, try another of the boot tapes you have just made until the "READY" message is displayed. Then, with the commands tape in the front deck, enter:

```
MIN ;AO:DRn
```

at the console where n is the drive number being "genned". The MIN program will be loaded from the disk into memory and will proceed to load the commands into the system and store them onto the disk. When the tape has been fully loaded and the messages "MULTIPLE IN COMPLETED" and "READY" are displayed, remove the front tape and proceed to load the second tape of commands. When all the commands have been loaded (usually three cassettes) the DOS generation procedure on the specified logical drive is complete.

Note that on some types of drive, notably the 9370-series ("Mass Storage") disk drives, each of the two logical disks on each disk pack must be DOSGENed individually (i.e. the DOSGEN

procedure must be done twice before the physical disk is completely DOSGENed). The second DOSGEN for such drives should be done after generating the boot tapes and before using MIN to load the commands.

After a disk has been fully DOSGENed in this manner, and if more than one disk drive is available, the faster DOSGEN command can be used to generate more disks. The use of the DOSGEN command is described later in this manual.

The Cassette DOS generation program can DOSGEN drives other than drive zero. In spite of this, it is important to recognize that the DOS must be resident on logical drive zero at an intermediate point in this DOSGEN procedure; therefore, the first DOSGEN done must be onto drive zero in order that a DOS be there when required. Subsequent DOSGENs can be onto any other drive, as long as drive zero then contains a fully DOSGENed disk.

CHAPTER 5. GENERAL COMMAND CHARACTERISTICS

Some features of the commands supplied with the DOS apply to most DOS commands. These characteristics and messages are discussed briefly in this chapter.

5.1 General Command Format

As mentioned in a previous chapter, DOS commands are entered as a command line. The general format of the command line is:

command [file spec][,file spec][,file spec]...[;options]

The item referred to as "command" is always required on a command line. This defines the command being issued to the system.

The items referred to as "file spec" represent one or more specifications for files. These files generally are input, output, scratch, or other files to be used by the command program. Usually the first such specifications represent input file(s), and the following specifications represent output or scratch file(s).

A square bracket convention is used here, as well as elsewhere throughout most Datapoint documentation, to indicate fields whose presence is optional. The corner bracket convention (as in <file spec>), represents replacement fields where the replacement field name is contained within the corner brackets. After the replacement is made, the corner brackets themselves do not appear in the resulting line.

The field indicated by "options", separated from the file specification fields by a semicolon, generally contains one or more option letters, which are defined for each specific command.

5.2 Signon Messages

Upon entering a system command, the command program being invoked will generally display a message identifying the command program. If the command is specific to one single DOS, the signon message will also identify which DOS the command is designed to execute under. The main purpose of the signon message is to allow the operator to determine, in the event of some difficulty, whether a superceded version of the command is in use.

5.3 Common Error Messages

Several error messages are common to many of the DOS commands. These error messages, and their meanings, include the following.

WRONG DOS. This message indicates that the version of the command program being run was intended to run on a specific version of the DOS, and that version is not the same as the DOS that is running. This message generally occurs either as a result of accidentally copying a command from one DOS to a different one, or attempting to use an obsolete version of a command under a newer DOS.

INVALID DRIVE. This message appears when one of the drive specifications given by the operator is invalid. Either the drive specification was not of the correct format, or the drive number specified exceeds the range available under the resident DOS.

NAME IN USE. This message occurs when the command's continued execution would necessarily result in a conflict of file name with an already existing file.

NAME REQUIRED. This message generally occurs when one of the file names required on the command line was not specified by the operator.

NO SUCH NAME. This message indicates that a file specified on the command line could not be found. Generally the name as specified is simply misspelled or otherwise incorrectly entered. However, sometimes this message will occur because the file desired is not in the current subdirectory (described later).

NO! THAT FILE IS PROTECTED. This message indicates that a request was made to modify a file that was write or delete protected.

WHAT? This message means that the command name (the first item on the command line being processed) is illegal. This usually indicates that either it is not a valid command, or that the command specified is not in the current subdirectory.

CHAPTER 6. APP COMMAND

6.1 Purpose

The APP command appends two object files together creating a third. Object files are files containing absolute object code in a format that can be loaded by the DOS loader. The transfer address of an object file is defined as the entry point of the program contained in the file.

6.2 Use

APP <file spec>,[<file spec>],<file spec>

The APP command appends the second object file after the first and puts the result into the third file. Note that neither if the input files are disturbed. If extensions are not supplied, ABS is assumed. The first two files (if a second is specified) must exist. If the third file does not already exist, it will be created. The first file's transfer address is discarded and the new file is terminated by the transfer address of the second file.

Omitting the second file specification causes the first file to be copied into the third file. For example:

APP DOG,,CAT

will copy the file DOG/ABS into the file CAT/ABS.

The first and third file specifications are required. If either is omitted the message

NAME REQUIRED

will be displayed. The second and third file specifications must not be the same.

Because the APP command recognizes the actual end of the object module contained in a file, APPing an object file, similiar to the example above, is one technique for releasing excessive unused space at the end of an object file.

CHAPTER 7. AUTO COMMAND

AUTO - Set Auto Execution

AUTO <file spec>

The AUTO command establishes the indicated program to be automatically executed upon the loading of DOS. If no extension is supplied, ABS is assumed. If there is already a file set for auto execution, the message

AUTO WAS SET TO NAME/EXTENSION (PFN).

will be displayed (where PFN is the physical file number). Regardless, the name specified will be recorded in the DOS table location reserved for the auto-execution information. A check is made to see if the file is an object file and if the file is on drive zero. If the specified file does not exist, the message

NO SUCH NAME

will be displayed. Note that if a program has been set to auto-execute, its execution can be inhibited by depression of the KEYBOARD key when DOS is loaded.

If no file spec is given in the commmand line, then the setting of the file to be auto-executed is not changed. However, if a file spec was present, then the message:

AUTO NOW SET TO NAME/EXTENSION (PFN).

will be displayed after the new auto-execution setting has been made.

If no <file spec> is entered and AUTO is not set, the message

NAME REQUIRED

will be displayed.

Note that the AUTO command does not make provision for file specifications to be given to the program which is to be automatically executed. This makes it impossible to use AUTO for programs requiring or accepting such parameters. AUTO also does not place anything in MCR$ (defined later). Therefore, programs

which use overlays with the same name (but different extension) as the program will not run. For more information, refer to the chapter describing the AUTOKEY command.

Auto-execution mode is cleared with the MANUAL command, described in a later chapter.

CHAPTER 8. AUTOKEY COMMAND

8.1 Introduction to AUTOKEY

Many users allow their Datapoint computers to run in an unattended mode. This allows large data processing tasks, perhaps running via the DOS command chaining facility (see CHAIN), to be run during the evening hours when no operator is present. (An example might be the creation of several new index files for one or more large, ISAM-accessed data bases). However, the momentary power failures which data processing users are being forced to contend with during times of shortage, thunderstorms and the like can bring down any computer not having special, uninterruptible power supplies. When this happens to a computer running in unattended mode, the office staff will generally return the next morning to find their computer sitting idle and its work unfinished.

The Datapoint computers are all equipped with an automatic-restart facility which can be used to cause them to automatically resume their processing tasks following such an interruption. The purpose of the AUTOKEY (and AUTO) commands is to provide a software mechanism for use by programmers who wish to handle such unusual circumstances and provide for the restarting of a processing task.

8.2 The Hardware Auto-Restart Facility

There are two little tabs on the back edge (the edge directly opposite from the edge where the tape is visible) of each cassette tape. The leftmost of these (as you look at the top side of the cassette) is the write protect tab, which prevents writing on the topmost side of the tape. The right-hand tab is the auto-restart tab.

Users who frequently use both sides of cassettes will probably immediately notice that if one turns over the tape, the assignments of these two tabs switch around, the tab which had been write protect now being auto restart and vice versa. This in fact is precisely what happens.

If the auto-restart tab on the rear cassette is punched out (or slid to the side on the newer cassettes), then the computer

will automatically re-boot, just like it does when RESTART is depressed, whenever a HALT instruction is executed. Assuming that the rear cassette drive contains a DOS boot tape, this will cause DOS to come up and give its familiar message, "READY".

Diskette 1100 users are provided with switch-selectable auto restart. The computer will either halt or automatically restart upon being stopped, depending upon the setting of an internal switch. This switch can be set by a Datapoint representative (SE or CE) upon request.

8.3 Automatic Program Execution Using AUTO

In order to provide a mechanism for programs to resume automatically following an interruption (such as a DATASHARE system, for instance, which might be running unattended) DOS has a comparable facility to enable a program to be automatically executed whenever DOS comes up. (Note that any loading and running the DOS, whether by an auto-restart, pressing the RESTART key, or under program control, will activate this facility).

The AUTO command is used to establish a program to receive control when DOS comes up. This setting can be cleared with the MANUAL command. For some applications, the AUTO and MANUAL commands are adequate to allow a programmed restart of a lengthy data processing task. However, some programs require parameters be specified on the command line, and these are obviously not present if no command line has been entered.

8.4 Auto-Restart Facilities Using AUTOKEY

AUTOKEY is simply a command program which can be AUTO'd. The way in which it works is very simple. If it is run via the DOS auto-restart facility, AUTOKEY supplies a command line just as if the same one line were entered at the system console. If AUTOKEY is run from the system console (or likewise from an active CHAIN), it simply displays the command line it is currently configured to supply and offers the user the option of changing that stored command line.

The command line supplied to AUTOKEY could do anything specifiable in one command line to the DOS; DATASHARE could be brought up, a SORT invoked, a user's own special restart program started or even a CHAIN begun. AUTOKEY, when used with AUTO, MANUAL, and CHAIN can therefore provide a very powerful facility.

8.5 A Simple Example

To specify a command line to be used during automatic system restart, simply enter:

```
AUTOKEY
```

at the system console. AUTOKEY will display a signon message and display the current autokey line if there is one. It then asks if this line is to be changed. If "N" is answered, AUTOKEY simply returns to the DOS and the familiar DOS "READY" message is displayed. If "Y" is answered, AUTOKEY requests the new command line to be configured and then returns to the DOS and "READY".

Alternatively, if the user wishes to simply specify a new command line to be configured regardless of the current setting of the AUTOKEY command line, he can merely place the new command line after the "AUTOKEY" that invokes the AUTOKEY command.

An example or two are in order. First, a simple one. Assume that XYZ Company has several of their sales offices on-line to their home office DATASHARE system, which is running completely unattended. Lightning strikes a powerline outside of XYZ Company's home office, and power is cut off for 15 seconds. As soon as power is restored, their Datapoint 5500 computer re-boots its DOS (since the right-hand tab on the boot tape has been punched out) and warmstarts the DATASHARE system. One command sequence to accomplish this would look like the following:

```
AUTOKEY
DOS.nn AUTOKEY COMMAND
NO AUTOKEY LINE CONFIGURED.
CHANGE THE AUTOKEY LINE? Y
ENTER NEW AUTOKEY LINE:
DS3
READY
AUTO AUTOKEY/CMD
AUTO NOW SET TO AUTOKEY/CMD (nnn)
READY
```

An alternate form of the above would be the following:

```
AUTOKEY DS3
DOS.nn AUTOKEY COMMAND
NO AUTOKEY LINE CONFIGURED.
ENTER NEW AUTOKEY LINE:
DS3  <--- (this is supplied automatically)
```

```
READY
AUTO AUTOKEY/CMD
AUTO NOW SET TO AUTOKEY/CMD (nnn)
READY
```

Once a program has been set for auto-execution, the only way one can bypass this is to hold down the KEYBOARD key while the DOS is coming up. This bypasses the auto-executed program and enters the normal command interpreter. The user then can use the MANUAL command to clear the auto-execution option.

8.6 A More Complicated Example

The following example uses many of the features of other facilities in the Datapoint system besides simply AUTOKEY. Explaining all of these in detail is beyond the scope of this section. The intention here is just to demonstrate the sophistication possible using AUTOKEY in conjunction with the other facilities within the DOS.

Let's assume that XYZ Company is running an eight-port Datashare system. Each of the company's seven sales offices around the country has a Datapoint 1100 computer which is connected up to the home office Datashare system as a port. (The eighth port is used by the home office's secretary, Susie, to maintain scoring for her bridge club.) During the day, each of the seven sales offices makes inquiries of the central inventory, price, and model code files through a system of Datashare programs, and another Datashare program lets them key orders into a file called "ORDERSn" where n is their port number. At the end of each business day, XYZ Company wants to process these orders. First they put the seven files all into one large file, sort it, and use a Datashare program to make corresponding entries into the master order file. The master order file is then reformatted and the index reconstructed. The final step is to create a second copy of the master order file onto magnetic tape, which will then be saved for backup purposes.

Since the operation just described is fairly lengthy, one of the more clever programmers at XYZ Company decided to allow it to run unattended after everyone else has gone home. They even set up Susie's MASTER program so that it automatically takes down the Datashare system and starts up the end-of-day processing one-half hour after the company's Los Angeles sales office (two time zones behind the Chicago main office) closes for the afternoon. When the daily processing is completed, Datashare is brought back up again so that it will be up by the time the first people start arriving at the New York sales office the next morning, an hour

before the Chicago main office opens.

In the event of an unanticipated power failure, the system will recover and bring itself back up, resuming operations at the last checkpoint established by AUTOKEY. Notice that the system is also left in a state such that after the chain completes, Datashare will automatically restart in the event of any possible system failure.

The following chain file ("OVERNITE/TXT") accomplishes the preceding, assuming that subdirectory "SYSTEM" is used throughout the chain. The chain file could be modified easily to eliminate this assumption. However, the chain file can be made almost arbitrarily complicated; the point here is simply to show one of many possible techniques for handling unattended operations which wish to restart automatically in the case of some failure. Notice that the chain file might have to be modified depending on the particular version of DSCON an installation is using.

```
// IFS S1
//. FIRST SET UP FOR AUTO RESTART IF REQUIRED.
AUTOKEY CHAIN OVERNITE;S1
AUTO AUTOKEY/CMD
//. NEXT APPEND TOGETHER THE SEVEN FILES.
SAPP ORDERS1,ORDERS2,SCRATCH
SAPP SCRATCH,ORDERS3,SCRATCH
SAPP SCRATCH,ORDERS4,SCRATCH
SAPP SCRATCH,ORDERS5,SCRATCH
SAPP SCRATCH,ORDERS6,SCRATCH
SAPP SCRATCH,ORDERS7,SCRATCH
//. NOW SCRATCH CONTAINS THE DAILY FILES.
AUTOKEY CHAIN OVERNITE;S2
// XIF
// IFS S1,S2
//. PHASE TWO SORTS FILE "SCRATCH" INTO "ORDERDAY".
SORT SCRATCH,ORDERDAY;1-5
//. NEXT CHECKPOINT HAVING BUILT "ORDERDAY".
AUTOKEY CHAIN OVERNITE;S3
// XIF
// IFS S1,S2,S3
//. PHASE THREE PROCESSES THE FILE WITH A DS3 PROGRAM.
DSCON
Y
N
Y
Y
1
DS3 PROCESS
```

```
//. THE MASTER ORDER FILE "ORDERMAS" NOW IS UPDATED.
AUTOKEY CHAIN OVERNITE;S4
// XIF
// IFS S1,S2,S3,S4
//. PHASE FOUR REFORMATS THE MASTER ORDER FILE.
REFORMAT ORDERMAS,SCRATCH:DR2;R
//. "SCRATCH" NOW IS A REFORMATTED COPY OF "ORDERMAS".
AUTOKEY CHAIN OVERNITE;S5
// XIF
// IFS S1,S2,S3,S4,S5
//. PHASE FIVE COPIES "SCRATCH" BACK TO "ORDERMAS"
COPY SCRATCH:DR2,ORDERMAS
//. "ORDERMAS" IS NOW READY FOR INDEXING.
AUTOKEY CHAIN OVERNITE;S6
// XIF
// IFS S1,S2,S3,S4,S5,S6
//. PHASE SIX RECREATES THE INDEX FOR "ORDERMAS"
INDEX ORDERMAS;1-16
//. THE INDEX HAS NOW BEEN REBUILT.
AUTOKEY CHAIN OVERNITE;S7
// XIF
// IFS S1,S2,S3,S4,S5,S6,S7
//. NOW DUMP MASTER FILE TO 9-TRACK MAGNETIC TAPE.
TAPE ORDERMAS/TXT;I/E
B
0
200X4
X
*
//. NOW THE BACKUP COPY OF "ORDERMAS" IS ON TAPE.
AUTOKEY CHAIN OVERNITE;S7
//XIF
//IFS S1,S2,S3,S4,S5,S6,S7
DSCON
Y
N
N
8
Y
AUTOKEY DS3
//. AND START UP DATASHARE FOR NEXT DAY.
DS 3
// XIF
```

8.7 Special Considerations

When building long chain files that allow for automatic restart, several perhaps obvious considerations must be made. Among these are that a file must not be changed in such a way that the change cannot be repeated if the previous checkpoint is actually used. To accomplish this, frequently the file being updated must be copied out to a scratch file, and the scratch file then updated. Following the completion of the update is when another checkpoint would be taken: following that the next phase would copy the updated file back over the original. Note that a checkpoint (i.e. resetting the AUTOKEY command line) would have to be before the creation of the dummy copy to be updated; putting a checkpoint between the creation of the copy to update and the actual updating process could result in the updating of a partially updated copy. A little thought when choosing places to update the AUTOKEY command line is called for to ensure that the chain may be resumed from any of them without incorrect results.

8.8 AUTOKEY and DATASHARE

Some users who make frequent use of the Datashare ROLLOUT feature will notice that AUTO-ing AUTOKEY with the AUTOKEY command line set to DSBACK will mean that whenever any port rolls out to any program or chain of programs, Datashare is automatically brought back up when that program or chain of programs finishes, regardless of whether or not DSBACK was included at the end of the port's chain file.

CHAPTER 9. BACKUP COMMAND

9.1 Purpose

The BACKUP command provides for making copies of entire DOS disks. The user can make either an exact mirror image copy of the input disk or can select reorganization, which will group files by extension and file name, remove unnecessary segmentation and allow deletion of unnecessary files. Reorganization also allows copying of DOS disks onto disks with locked out cylinders that differ from those on the input disk. Some special considerations apply for specific disk configurations; these considerations, if any, are discussed in the System Guide for the specific DOS being used.

9.2 Use

A disk backup is initiated by the operator entering the following command:

BACKUP <input drive>,<output drive>

Input drive and output drive are specified as :DRn or :Dn. The drive selected as the INPUT DRIVE MUST BE WRITE PROTECTED; that is, it must be in "read only" mode or have its "protect" light on for 9370 and 9350 series drives respectively. The requirement for the input drive to be write protected is absent on the 9380 series flexible diskettes. The program will respond by displaying the message:

DRIVE n SCRATCH?

If the disk on drive n is scratch (note that BACKUP deals with logical drives), enter a "Y" . Any other reply will cause the program to return to DOS. If you do reply "Y", the program will display the message:

ARE YOU SURE?

If you are absolutely sure that you want to write over the output disk, type "Y" again and press the enter key. Any other reply will cause the program to return to DOS. If the output (logical) disk has not been DOSGENed or the DOS file structure on it has been damaged, the message:

DOSGEN YOUR DISK FIRST

will appear and control returns to DOS. If the output (logical) disk has been DOSGENed and seems in reasonable shape, the following message is displayed:

FILE REORGANIZATION?

Note that the option to reorganize during the copy is mandatory if the output disk has any bad cylinders on it locked out. If this is the case, the "FILE REORGANIZATION?" question is bypassed completely and reorganization is assumed.

If you wish to reorganize the files being transferred to the output disk, enter a "Y" in response to the reorganization question. In this case, see the section on reorganizing files for further instructions.

If you do not wish to reorganize your files and desire a mirror image copy of your input disk, enter an "N" in response to the reorganization question.

9.3 Mirror Image Copy

If you have typed "N" in response to the file reorganization question, the program will ask the question:

DO YOU WANT TO COPY UNALLOCATED CLUSTERS?

Type "Y" and press the enter key if you want all data on the disk copied regardless of whether or not it is in an area allocated by DOS. This option is preferred in cases where you suspect that your DOS files may be partially destroyed or the output disk has never been fully initialized with data.

Type "N" and press the enter key if you wish to copy your disk as quickly as possible without copying unused areas of the input disk. "Y" and "N" are the only replies allowed!

9.4 Reorganizing Files

If you have typed "Y" in response to the file reorganization question, the program will copy the System files, sort the directory names, and allow the operator to delete files before copying the files to the disk copy.

9.4.1 Copying DOS to Output Disk

Various program status messages will appear during the copying of DOS. System tables are initialized and then the SYSTEMn/SYS files are copied to the output disk.

9.4.2 Deleting Named Files

When all directory names have been sorted into file extension followed by file name sequence and all unnamed files have been copied, the following question will be displayed:

DELETE ANY FILES DURING REORGANIZATION?

Type "N" and press the enter key if all files are to be copied. Type "Y" and press the enter key if you wish to delete any files. If you reply "Y" a message asking which files are NOT to be copied will appear. The lower screen will be filled by a numbered list of files for you to choose from. Type the number or range of numbers (nn or nn-nn) found next to names of individual files you wish deleted. Type "ALL" and press the enter key if wish to delete all of the files in the list. The files selected for deletion will be erased from the list. When all desired deletions have been made from a list, type "." and press the enter key to advance to the next list of file names.

When all file name lists have been examined, the program will advance to the copy named files phase.

9.4.3 Copying Named Files

Files with names in the system directory are copied in alphameric file extension, file name sequence. The name of each file is displayed as it is copied. All files are written as close together as possible with an absolute minimum of segmentation.

9.5 Use of KEYBOARD and DISPLAY Keys

The KEYBOARD and DISPLAY keys may be pressed any time messages are being displayed. Depressing the DISPLAY key will hold the current display until the key is released. Depressing the KEYBOARD key will cause the program to terminate and return to DOS.

9.6 Error Messages

During the execution of BACKUP the following error messages may appear:

*** PLEASE PROTECT YOUR INPUT DISK ***

Action: Write-disable the input drive.

INVALID DRIVE SPECIFICATION!

Action: Retype the BACKUP command with correct <input-drive> and <output-drive> specification.

ILLEGAL OUTPUT DRIVE!

Action: <input-drive> and <output-drive> have been specified as the same drive! Retype BACKUP command with correct specification.

BAD CLUSTER ALLOC TABLE!

Action: A bad Cluster Allocation Table has been detected on the input disk. The Cluster Allocation Table may be able to be fixed using the REPAIR command.

CYLINDER 0 OF BACKUP DISK IS UNUSABLE!

Action: Your scratch disk cannot be used for a system disk due to surface defects in cylinder 0. Use another output disk and start over.

SYSTEMn /SYS IS MISSING!

Action: Your DOS disk cannot be reorganized due to a missing system file. Catalog the missing system file on your input disk and start over.

PARITY- :DRn address

Action: An irrecoverable parity error has been detected on drive n during the BACKUP operation. The address is shown for each error. If drive n is your output disk, DOSGEN must be rerun to lockout the bad addresses or use a different scratch disk for mirror image copy. If drive n is your input disk, new parity will be computed and the record will be copied. Note the error address and check for errors when copy is complete.

9.7 Reorganizing Files for Faster Processing

After a DOS disk has been used for awhile, the file structure becomes fragmented and related files become scattered. The more the disk is used the more total system performance is degraded due to increased disk access time. System degradation is especially noticeable when DATASHARE is being used. File reorganization using the BACKUP program is one way to clean up DOS disks and improve their efficiency.

BACKUP reorganization improves system efficiency by making the following changes:

- File segments are consolidated
- Files are packed more closely together
- Related files are clustered together
- Unused trash files are removed (optionally)
- Files are rewritten reducing marginal parity errors

Care should be exercised in naming files so that related files have the same file extensions and similar file names that will allow them to be grouped when the system directory is sorted.

9.8 BACKUP with CHAIN

Because BACKUP requires that its input drives be write protected, does not abort if parity errors occur during the backup, and may ask different questions depending upon the condition of the input and output disks, BACKUP generally should not be invoked from a CHAIN. Since the BACKUP operation is so critical to the protection of important files, an operator should monitor the entire backup operation.

9.9 Clicks during Copying

On some versions of BACKUP, generally those intended to run on smaller disks, a click occurs each time an unwritten sector is copied (reorganization mode only). A file which when copied results in a lot of clicks (more than a dozen, perhaps) can probably be reduced in size, without any data loss, by using APP or SAPP as appropriate. Clicks normally occur only at the end of a file. Only in physically random accessed files should clicks

occur in the middle.

CHAPTER 10. BLOKEDIT COMMAND

10.1 Purpose

The BLOKEDIT command provides for DOS text file manipulation. The command copies lines of text from any DOS text file(s) to create a new text file.

The BLOKEDIT command is useful for such things as:

New program source file generation by copying routines from existing program source files;

Existing program source file re-arranging by copying the lines of source-code into a new sequence (into a new source file);

Re-arranging lines or paragraphs of a SCRIBE file into a new file.

In this Chapter, the following applies:

Text file means a DOS EDIT-compatible file.

Line means one line of a text file as displayed by the DOS LIST program.

10.2 File Descriptions

BLOKEDIT deals only with text files. For any given application there will be one text file called the COMMAND FILE which will hold the controlling commands for BLOKEDIT, there will be one or more text files called SOURCE FILES from which lines of text will be copied, and there will be one text file called the NEW FILE which will be the desired end result for the application.

10.2.1 Command File

The command file is the controlling factor in BLOKEDIT execution. The command file specifies which source files will be used and which lines of text will be copied from them. A command file must be created by the DOS EDIT program before BLOKEDIT can be used.

There are three kinds of lines that are meaningful in a command file: COMMENT lines, COMMAND lines, and QUOTED lines.

A COMMENT line is a line which has a first character of period.

This is an example of COMMENT LINES:

```
.
. THESE THREE LINES ARE COMMENT LINES.
.
```

As in program source files, a comment line may have explanatory notes or nothing at all following the period.

A COMMAND LINE is a line which has a SOURCE FILE NAME and/or source file LINE NUMBERS, or begins with a double quote symbol (").

The following are some example command lines:

```
FILENAME/EXT:DRO          NAME THE SOURCE FILE
1-100                     COPY LINES 1 THRU 100
350-377                   COPY LINES 350 THRU 377
```

A command line must have a first character of an upper-case alphabetic character, or a digit, or a double quote symbol.

A command line that begins with an upper-case alphabetic character indicates that a new SOURCE FILE is being named. A new source file can be named only by putting the name of the file at the very beginning of the command line. Optionally, the extension and/or drive number for the file may be included with the source file name.

A command line that begins with a digit indicates that the command line will have one or more numbers, which are the numbers of the lines to be copied from the source file previously specified in the command file into the new file.

A command line that begins with a double quote symbol indicates the beginning/ending of QUOTED LINES. The only information used by BLOKEDIT in a command line that begins with a (") is the (") itself, therefore the rest of the line can be used for comments.

A QUOTED LINE is a line between a pair of command lines which begin with a double quote symbol.

This is an example of QUOTED LINES:

```
" THIS IS THE BEGINNING OF QUOTED LINES COMMAND LINE.
INCMNT  HL    COUNT           POINT TO COUNTER
        LAM                   LOAD TO "A" REGISTER
        AD    1               INCREMENT BY 1
        LMA                   RESTORE TO MEMORY
" THIS IS THE ENDING OF QUOTED LINES COMMAND LINE.
```

There may be more than one quoted line between the command lines that begin with (") . A quoted line will be copied directly from the command file to the new file. Quoted lines enable a person BLOKEDIT user to include original lines of text in a new file along with lines copied from source files.

10.2.2 Source File

The SOURCE FILE is a DOS EDIT-compatible text file from which lines will be copied. source files are named in the command file for a BLOKEDIT application, and the lines to be copied from the source file will also be specified in the command file. It will be useful to have a listing of a source file with line numbers, as produced by the LIST command, when creating the command file for a BLOKEDIT application.

10.2.3 New File

The NEW FILE is a DOS EDIT-compatible text file produced by the BLOKEDIT command. The new file is named at BLOKEDIT execution time by the second file specification entered on the command line (see below).

10.3 Using BLOKEDIT

Before the BLOKEDIT command can be used one must create a command file as described above. When the BLOKEDIT command is to be executed, the operator must enter the following command line:

```
BLOKEDIT <file spec>,<file spec>
```

The first file specification refers to the command file and the second file specification names the new (output) file. If no extension is supplied with the first file specification, TXT is assumed. If no extension is supplied with the second file specification, the extension given or assumed for the first file is used. If no drive is given for the first file, all drives are searched. If no drive is given for the second file, the drive given or assumed for the first file is used. The specified output file must not exist on any drive on line.

10.3.1 Messages

This section describes the operator messages that BLOKEDIT may display on the CRT screen during execution. Some of the messages are monitor messages to keep the operator informed of the progress of the program, while other messages are error messages.

PROCESSING COMMAND LINE .. CURRENT SOURCE FILE IS ../..:DR.

This message is the BLOKEDIT monitor message. This message is displayed while BLOKEDIT is writing lines of text to the new file. The monitor message displays the command file line number currently being processed and the name, extension, and drive number of the last named source file.

If BLOKEDIT detects an error in the command file the monitor message is rolled up the screen one line, an appropriate error message is displayed, and the monitor message is re-displayed. In this way a screen-log of where errors in the command file occur is maintained.

COMMAND AND NEW FILE NAMES REQUIRED.

This message is displayed if the operator did not name both a and a new file when the BLOKEDIT command was called.

COMMAND FILE DRIVE INVALID.

This message is displayed if the operator specified for the command file a drive number that is invalid.

NEW FILE DRIVE INVALID.

This message is displayed if the operator specified for the new file a drive number that is invalid.

COMMAND AND NEW FILE NAMES MUST NOT BE IDENTICAL.

This message is displayed if the operator specified command file and new file names the same and the extension and the drives for the files were specified or assumed to be the same. Defaulting of extensions and drives is described in an earlier paragraph.

COMMAND FILE NOT FOUND.

This message is displayed if the command file name was not found on the drive(s) specified or assumed.

NAME IN USE.

This message is displayed if the specified output file was found on the drive(s) specified or assumed. BLOKEDIT will not write into an existing file.

BAD FILE SPECIFICATION.

This message is displayed if the first character of a command file line other than a quoted line is an upper-case alpha character but the DOS file specification was not recognizeable.

SOURCE FILE NOT FOUND.

This message is displayed if the source file specified could not be found. It is probably either misspelled or in a different subdirectory.

BAD LINE NUMBER SPECIFICATION.

This message is displayed if a command file line other than a quoted line began with a digit but contained an unrecognizeable line number specification.

Here are some examples of valid line numbers:

4	A single digit is acceptable.
999999	A line number may have up to six digits.
100-364	Two numbers may be separated by a "-".
34,55-78,100-147	Commas may separate numbers.

Here are some examples of invalid line numbers:

1A	Only "-", ",", or space after a digit.
1234567	Number has more than six digits.
17-34-77	Only two numbers separated by "-".

LINE NUMBER ZERO IGNORED.

This message is displayed if a line number of zero is specifed in a command file line.

START LINE NO. > END LINE NO., IGNORED.

This message is displayed if the first number of a line number pair is larger than the second number of the pair, as in: 235-176.

BAD DATA IN SOURCE FILE LINE .....

This message is displayed if BLOKEDIT discovers non-ASCII characters in a source file. The line number will be displayed following the message.

SOURCE FILE WENT TO E.O.F..

This message is displayed if the source file from which lines were being copied ended before the specified lines were finished.

TEXT TRANSFER DONE.
NEW FILE'S LINE COUNT IS .....

This message is displayed when all of the command file lines have been executed. The number of lines in the new file is displayed following the second line.

CHAPTER 11. BOOTMAKE COMMAND

BOOTMAKE - Generate a DOS bootstrap cassette

The BOOTMAKE command writes a DOS bootblock onto the cassette tape in the front tape deck. BOOTMAKE accepts no operands. To use it, simply enter:

```
BOOTMAKE
```

at the system console. The command asks if the cassette in the front deck is scratch. If it is, the tape is rewound and a DOS bootblock written onto it.

The BOOTMAKE command then rereads the bootblock to insure that the cassette is good. In addition, the bootblock checks its own parity immediately upon loading and halts if it finds it has not been loaded properly.

If the machine halts upon booting repeatedly and other boot tapes work on the same machine, then the boot tape which causes the boot operation to halt is not a good tape and should not be used.

CHAPTER 12. BUILD COMMAND

12.1 Purpose

It is occasionally desirable to be able to create a text file without having to use the standard DOS editor. It is particularly useful to be able to generate a text file comprised of lines contained within a CHAIN file (to be described in a later chapter), possibly selected by options on the CHAIN command line. BUILD offers precisely this facility.

BUILD is also useful for rapid generation of very short text files, such as two and three line CHAIN files or BLOKEDIT command files.

12.2 Use

The BUILD command is invoked by entering the command line:

BUILD <file spec>[;<end character>]

The file specification defines the output file. This output file specification is always required. If the named file does not exist, it is created.

The end character is optional. If no end character is specified on the command line, BUILD terminates upon receiving a null input line (a null input line is a line consisting of only an ENTER. A blank line is not a null line). Note that neither the EDIT command nor the BLOKEDIT command normally ever write a null line to a disk file, nor does the CHAIN command, (described in a later chapter), normally respond to a KEYIN request with a null input line. Specifying an end character, therefore, allows an end-of-input character to be defined to BUILD so that it can be invoked from within a CHAIN file.

BUILD accepts input lines from the keyboard and writes each one to the output file. When BUILD is ready to accept an input line it displays a colon (:) as a prompting character. Each input line BUILD receives is tested for the presence of the specified end character, if any, as the first character entered. If the end character is present as the first character of the entered line, the end line is discarded (it is not written to the output file),

and an end of file mark is written to the output file and the output file closed by returning to DOS.

12.3 A Simple Example

Suppose that the operator wishes to construct a simple CHAIN file to establish a program to be auto-executed, so that the auto-execute request can be accomplished later with a single command line entered at the keyboard. All that is required is to enter at the system console:

```
BUILD <chain file spec>;*
AUTOKEY <program name>
AUTO AUTOKEY/CMD
*
```

Upon receiving the "*" input line, BUILD closes the output file and terminates. Note that in the two places where the "*" appears, any enterable character could have been used. (This allows nesting calls to BUILD, which can be very useful in the BUILDing of chain files). After the BUILD command is finished, the output file named on the BUILD command line contains the following two lines:

```
AUTOKEY <program name>
AUTO AUTOKEY/CMD
```

12.4 A More Sophisticated Example

Many Datashare users have a need for a MASTER program for each separate port they have on line. Generally, these eight or sixteen MASTER programs are identical except for a port number or other (relatively short) port-specific information.

One way of maintaining these multiple MASTER programs is to create many copies of the same program and then use the EDIT command (described later) to modify each of the many source files as necessary. However, this makes subsequent changes to the MASTER program tedious since many different copies have to be updated in parallel, and the possibilities for errors should be obvious.

A much easier solution for the problem is to make the port number definition an <u>inclusion</u>, and to BUILD this inclusion file in the chain file used to compile the multiple programs. A portion of the resulting chain file might look as follows:

```
BUILD PORTNUM;*
```

```
PORTN FORM "01"
*
DBCMP MASTER,MASTER1
BUILD PORTNUM;$
PORTN FORM "02"
$
DBCMP MASTER,MASTER2
BUILD PORTNUM;&
PORTN FORM "03"
&
DBCMP MASTER,MASTER3
 etc.
```

By then simply specifying an INCLUDE statement in the MASTER program, it can easily be arranged to have each different MASTER object program have a different port number. Notice that in the example above, the file PORTNUM built by BUILD could be of any length; in this example it is only one line long since that is all that is required for this example. In the case used above, building the inclusion file with BUILD results in only requiring two source files (instead of maybe sixteen) to support many ports, and one of those source files is transient (PORTNUM/TXT) and can be deleted at the end of the chain. Changes to the MASTER program can be accomplished by simply modifying one version of the program and then running the chain file to create all the required object programs.

It is also possible, through BUILD nesting, to create chain files which during execution of the chain construct other chain files and execute them automatically upon completion of the first chain (since the last statement of a chain file is allowed to be a CHAIN command).

The references to CHAIN made here may be premature, since CHAIN is discussed in a later chapter, but are included because BUILD and CHAIN can be of great usefulness when used together in this manner. With a little imagination it is genuinely surprising to discover the level of sophisticated job and program control that the combination of these two very useful programs can provide.

CHAPTER 13. CAT COMMAND

13.1 Purpose

The CAT command selectively displays DOS filenames from the directory. One may choose to display all catalogued filenames on all drives online or specific filenames on specific drives.

13.2 Use

The CAT command is invoked by entering the command line:

CAT [<name>][</ext>][:DR<n>][,L]

where: <name> specifies the filename or a portion of the filename, <ext> specifies the extension or a portion of the extension, <n> specifies the logical disk drive number, and L specifies list only those files in the current subdirectory.

Directory entries are displayed in the form:

NAME/EXTENSION (PFN)P

where PFN is the physical file number in octal (0-0377) and P is the protection on the file (D for deletion, W for write, and blank for none). If the file displayed is in a subdirectory other than system, the directory entry is displayed in the form

NAME/EXTENSION-(PFN)P

with the dash indicating a subdirectory entry. All drives are searched, unless a specific drive is requested, and as each drive is scanned, the line

---- DRIVE n (subdirectory name):

is displayed. (This line is not displayed if the drive is not on line, or if no files from it are to be displayed).

Depressing the DISPLAY key causes the catalog display to pause as long as the key is held. Depressing the KEYBOARD key causes the catalog display to terminate. If the CAT command is parameterized by only an extension, only files of that extension

will be displayed. If the CAT command is parameterized by only a name, only files of that name (all extensions) will be displayed. If the CAT command is parameterized by a name and an extension, only files of that root name and extension (all drives) will be displayed. If the CAT command is parameterized by only the drive number, only files on that drive will be displayed. If only a portion of the filename is entered, all files beginning with the letters specified will be displayed. For example, entering:

```
CAT /T
```

would cause the display of all files on all on-line drives whose extensions start with "/T". Entering:

```
CAT MA:DR2
```

would cause the display of all files on drive two whose file names start with "MA".

CHAPTER 14. CHAIN COMMAND

14.1 Introduction

The CHAIN command enables an operator to employ a disk file in defining job procedures in terms of any sequence of other DOS programs. This file can also supply parameters to the DOS programs invoked, allowing automatic control of execution options. For example, CHAIN could be used to run the SORT utility on several files and then to print listings of these files. The headings on the listings could contain a date which was entered as a parameter to the CHAIN command. Another common use of CHAIN includes program generation of large systems where one must often execute a number of assemblies, create a complex of files by appending together combinations of the object files created by the assemblies, and then make an LGO tape containing the combined files. In this application, one usually wants to be able to enter the date to be printed in each assembly listing heading, or to restrict assembly to only a subset of the entire system of programs. This can all be performed by the CHAIN program.

Basically, CHAIN replaces the DOS keyboard entry routine with a routine which reads a line from a file each time the keyboard entry routine is called. Therefore, each time any program would normally request a line to be entered from the keyboard it will get the line from the file. The DOS program has no idea that the line is coming from the file instead of from the keyboard.

CHAIN only replaces the DOS keyboard entry routine (KEYIN$). Therefore, only programs that use this routine for input will receive their input from the chain file. Programs which have their own input routines, like the Editor, can be invoked from a chain file but editing must be done manually by the operator. The CHAIN program itself cannot be called from within a CHAIN file unless the CHAIN command is the last command in that chain. If CHAINing with a CHAIN file (recursion) is attempted, an error message is given and the chain is aborted. The chain is also aborted when a CHAIN-invoked program makes an exit to DOS that implies that an error of some kind has been made. The error message given by the program will generally remain on the screen after the chain is aborted.

In a sense, CHAIN is a macro facility. It allows the

operator to define a macro procedure under a name (the CHAIN file name) in terms of other smaller operations (other DOS programs). This procedure can then be invoked by simply giving the name of the procedure file to the CHAIN program. The additional facilities of decision making within the procedure through conditional statements (including logical operations and micro substitution of strings from the CHAIN command line) allow an almost unlimited extension of this concept.

14.2 Simple Use of CHAIN

As mentioned in the introduction, CHAIN really does nothing more than simulate the operator when the standard DOS keyboard entry routine is called. The data to be entered is obtained from a file whose name is given on the same line as the CHAIN command. For example, one could edit a file called APPFILES using the standard DOS Editor. It could contain the following lines:

```
APP DWS,DSTATH,DS/OV1
APP DS/OV1,DSKED,DS/OV1
APP DS/OV1,DINT,DS/OV1
APP DS/OV1,DINIT,DS/OV1
```

When one then entered the DOS command:

```
CHAIN APPFILES
```

the above lines would be entered in the sequence shown whenever the DOS requested another command line. The conceptual simplification and reduction in probability of operator error is obvious. Instead of having to remember the names of all the files each time the file DS/OV1 is to be created, the operator need only remember the file name APPFILES.

When the last line of the CHAIN file has been exhausted, and a new DOS command is desired by the system, the DOS is reloaded and commands are again accepted from the operator at the keyboard. However, if the file is exhausted while a program is requesting data, the CHAIN ABORTED! message is given and the program currently being executed is abandoned.

Although the above example showed only DOS commands being given by the CHAIN program, it should be remembered that all keyboard entries requested through the standard DOS keyboard entry routine will obtain their data from the CHAIN file. For example, if in the above example the files not needed after the appending had taken place were to be killed, the file would have contained the following additional lines:

```
KILL DWS/ABS
Y
KILL DSTATH/ABS
Y
KILL DSKED/ABS
Y
KILL DINT/ABS
Y
KILL DINIT/ABS
Y
```

Note that the 'Y' given after each KILL command is not another DOS command but is the response to the message 'ARE YOU SURE?' that the KILL command displays. Another example would be one of an assembly. When obtaining listings, the assembler requests the entry of a heading. Thus, the CHAIN file would need to contain the heading as well as the assembly command:

```
ASM TSDWS;XL
DATASHARE 2.1 WORKING STORAGE
```

Some DOS programs can go through a rather complex set of requests for input which can make them hard to use with the CHAIN program without making a mistake. For example, the old DOSASM4 used to ask for program options one at a time with a message for each. This is a nice feature if one is entering the data from the keyboard but, on the other hand, makes it almost impossible to use the program with CHAIN without making an error in the procedure file. The problem is aggravated by the fact that the number of options requested varies depending upon the response to certain options. For this reason, most DOS programs allow almost all options to be specified on the command line and keep the variation in the number of keyin requests to a minimum. It is good practice for all DOS programs to be written with this in mind to facilitate their use with CHAIN.

An additional item to keep in mind is the fact that some DOS programs use their own keyboard entry routine as well as the one provided by the DOS. This enables the program to avoid the use of the CHAIN procedural lines when special operator intervention is required. An example is the TAPE program (industry compatible 9-track tape handler) which requests operator intervention if the end of the reel of tape is reached while a file is being written. In this case, the program asks if the writing is to be continued on another reel of tape and if so waits for the operator to change reels. After the operator indicates to the program that the reel has been changed, the program can continue writing files, getting the names from the CHAIN procedural file. If the TAPE program had

used the standard DOS keyboard entry routine, the file names would have been given for reel change responses and, needless to say, the order of program execution would have been incorrect.

14.3 More Advanced Use of CHAIN

14.3.1 Tag definition

The CHAIN command line can be parameterized by tags given after the procedural file specification, if the file specification is followed by a semicolon. Tags may be from one to eight characters in length and have values from zero to eight characters in length. A tag must start with a letter and contain only letters or digits. A tag's value can contain any character except the <#> symbol. A tag is given a non-null (non zero length) value by enclosing the value to be assigned between <#> symbols. For example, in

```
CHAIN MAKETEST;LIST,DAY#17#,TIME#02:30#
```

the tag LIST has a null value, the tag DAY has the value 17, and the tag TIME has the value 02:30. CHAIN allows two uses to be made of tags. Tests can be made to determine whether a tag of a given name has been entered on the CHAIN command line and the value of a tag can be substituted for part of a line within the procedural file.

14.3.2 Phases of execution

CHAIN executes a compilation phase and an execution phase. In the compilation phase (CHAIN/CMD), the specified procedural file is read and a new file is created (always named CHAIN/SYS) and deleted at the successful completion of the procedure. This new file consists of only the lines to be used in the particular procedure to be executed. All compilation comment lines and conditional items not to be used are eliminated, all substitutions of tag values are made where indicated, and the space compression in the file is eliminated to make it easier on the routine that overlays the DOS KEYIN$.

The execution phase (CHAIN/OV1) overlays the DOS KEYIN$ with a routine which fits in the same space and appears the same upon exit (the HL and DE registers are the same as if data had been entered from the keyboard and the display and cursor positioning on the screen is the same) except the input line is obtained from the CHAIN/SYS file instead of from the keyboard. If the line read

is longer than the maximum specified by the calling program, the chain is aborted.

After the KEYIN$ routine is overlayed, the execution phase reads in one line and supplies it to the DOS as if the line had been entered as a DOS command. If this produces an execution time comment line, the line is simply displayed and another line is read (which may also be an execution comment). Execution time comments begin with "//." as the first three characters in the line. There is a variation on the execution time comment line, which is called an operator break point. Operator break point lines begin with "//*" as the first three characters in the line. This line is displayed as in an execution time comment except afterward the chain waits for an operator to depress the KEYBOARD or DISPLAY key. After one or the other of these keys is depressed, the chain continues to the next line as in an execution time comment.

If the end of the CHAIN/SYS file is reached while a DOS command is being sought, the chain is determined to be finished. At this point the CHAIN/SYS file is deleted and DOS is reloaded (restoring the KEYIN$ routine to its normal state).

When a program terminates by jumping to the DOS EXIT$ entry point or to the label NXTCMD in the command interpreter, the routine overlaying KEYIN$ loads and executes the CHAIN/OV1 file which reloads DOS, restores the CHAIN KEYIN$ routine, and then reads the CHAIN/SYS file for the next command.

14.3.3 Tag existence testing

Thoughout this subsection, the term "operator" refers to an operator in the expression evaluation sense, like a "+". It should not be confused with the human operator at the system console.

CHAIN contains an IF operator which allows a test to be made for the existence of one or more tags in the CHAIN command line. If the test proves positive, then the lines following the IF operator will be included in the CHAIN/SYS file. If the test proves negative, then the lines will not be included in the CHAIN/SYS file. This will hold true until either the ELSE or XIF operators discussed below are reached. All CHAIN operators are denoted by a // as the first two characters in the line. If the third character is additionally a period, then an execution time comment is indicated. Otherwise, any number of intervening spaces (including zero) are scanned until an operator is reached. If the operator scanned is not one of the defined operators, an error

message will be given and the procedure aborted. The examples all show one space between the two slashes and the operator in order to make them more readable.

The IF operator has two variations, IFS and IFC, which stand for if-set and if-clear. The IFS operator proves positive if any of the tags listed exist. For example:

```
// IFS FLAG1
```

will prove positive if FLAG1 was mentioned in the CHAIN command line. The IFC operator proves positive if any of the tags listed were not mentioned. For example:

```
// IFC FLAG2
```

will prove positive if FLAG2 was not mentioned in the CHAIN command line. One can test to see if all of a group of tags exist by having multiple IF statements in sequence. For example:

```
// IFS FLAG1
// IFS FLAG2
// IFS FLAG7
```

will allow the following lines to be used in the procedure if FLAG1 is set AND if FLAG2 is also set AND if FLAG7 is also set. A comma between tag names on an IF line actually performs the logical OR function. A logical AND function may be performed by putting a period between tag names. For example:

```
// IFS FLAG1.FLAG2.FLAG7
```

is equivalent to the above example.

```
// IFS FLAG1.FLAG2,FLAG3,FLAG4.FLAG5.FLAG6
```

will allow the following lines to be used in the procedure if FLAG1 and FLAG2 exists or if FLAG3 exists or if FLAG4 and FLAG5 and FLAG6 exists. Note that the IF operators are scanned only if procedure lines are being used. Thus, if one of the IF operators has proven negative and has inhibited the use of procedure lines, all following IF operators will be ignored until either an ELSE or XIF operator is reached.

As mentioned in the first paragraph of this section, the two CHAIN operators that can cause procedure lines to be put back into use are ELSE, which reverses whether or not the procedure lines are being used, and XIF, which unconditionally turns on the usage

of procedure lines. For example, if the CHAIN file MAKETEST contained the following:

```
// IFS LIST
ASM TEST;XL
TEST PROGRAM
// ELSE
ASM TEST
// XIF
APP DATA,TEST,TEST/CMD
```

and the CHAIN command was given as follows:

```
CHAIN MAKETEST;LIST
```

then the procedure followed would be:

```
ASM TEST;XL
TEST PROGRAM
APP DATA,TEST,TEST/CMD
```

However, if the chain command was given as follows:

```
CHAIN MAKETEST
```

then the procedure followed would be:

```
ASM TEST
APP DATA,TEST,TEST/CMD
```

ELSE and XIF can only be inhibited by the use of the BEGIN and END operators discussed in the Section on Additional CHAIN Operators.

14.3.4 Comment lines

CHAIN allows for two types of comment lines within the procedural file. One type already mentioned is the execution time comment. This type may appear only before a DOS command entry and will not appear until just before that command is to be executed. For example, the procedure file containing:

```
//. ASSEMBLY OF THE TEST PROGRAM
ASM TEST;XL
TEST PROGRAM
```

would cause the first line to be displayed before the assembly was executed. A variation on the execution time comment is the operator break point. For example, the procedure file containing:

```
//* INSERT TAPE Z12548 INTO THE FRONT CASSETTE DECK
MOUT ;LV
TEST
DATA/TXT
*
```

would cause a BEEP and the first line to be displayed. At this point the machine would wait for the operator to depress either the KEYBOARD or DISPLAY key and then continue with the MOUT process.

The second type of comment line is a compilation time comment. This line is not included in the procedure but is displayed on the screen immediately after it is read from the procedural file. This is useful in communicating to the operator what procedure is about to be followed by CHAIN.

Both types of comment lines will be ignored (not displayed or written) just as other procedure lines if a test has proven negative and an ELSE or XIF operator has not been reached. For example, if the following procedure file MAKETEST was created:

```
. ASSEMBLY OF TEST PROGRAM
// IFS LIST
. YOU ARE GOING TO GET A LISTING
ASM TEST;XL
TEST PROGRAM
// ELSE
. YOU AREN'T GOING TO GET A LISTING
ASM TEST
```

and the CHAIN command:

```
CHAIN MAKELIST;LIST
```

was given, then only the lines:

```
. ASSEMBLY OF TEST PROGRAM
. YOU ARE GOING TO GET A LISTING
```

will appear on the screen before the procedure is executed. If, however, the CHAIN command:

```
CHAIN MAKETEST
```

was given, then only the lines:

```
. ASSEMBLY OF TEST PROGRAM
. YOU AREN'T GOING TO GET A LISTING
```

will appear on the screen before the procedure is executed.

14.3.5 Tag value substitution

So far in the discussion the value of a tag has not been used. Note that the existence of a tag can be tested regardless of its value. Thus the procedure file can test to see if any tags with values have been forgotten by the operator. A tag value is simply substituted wherever a pair of "#" symbols are found with a syntactically valid tag name between them. An example will eliminate a large number of words. Assume that the procedural file MAKETEST is as follows:

```
ASM TEST;XL
TEST PROGRAM ASSEMBLED ON #DATE#
ASM ASDF;L
THE ##FLAG1## PROGRAM #FLAG2#
//. ##FLAG1##FLAG3#FLAG2#
```

A CHAIN command of:

```
CHAIN MAKETEST
```

would produce a CHAIN/SYS file that looked exactly the same since if the tag between "#" symbols does not exist, then no substitution at all is performed. However, a CHAIN command of:

```
CHAIN MAKETEST;DATE#17JAN74#,FLAG1#FLAG4#,FLAG2#ZXCV#
```

would produce a CHAIN/SYS file as follows:

```
ASM TEST;XL
TEST PROGRAM ASSEMBLED ON 17JAN74
ASM ASDF;L
THE #FLAG4# PROGRAM ZXCV
//. #FLAG4#FLAG3#FLAG2#
```

Observe, for a moment, how the "#" symbols were handled in the next to the last line. The first two "#" symbols did not enclose a syntactically valid tag name. Therefore, the first "#" was simply passed through and a pairing "#" for the second "#" was sought. This was found and a syntatically valid name (FLAG1) was found to be between. So the value of FLAG1 was substituted for the characters #FLAG1# in the line and then the scan continued.

The following pair of "#" enclosing the word PROGRAM was not used as a tag name because the word PROGRAM was terminated by a space. To be used for a tag name, the substitution specification must be terminated by the "#" symbol.

Now observe the last line in the example above. The first three "#" symbols were handled in the same way as for the next to the last line. However, #FLAG3# did make up a syntatically valid substitution specification so it was used. FLAG3 did not exist so #FLAG3# was simply passed through. At this point however, the only characters remaining to be scanned were FLAG2# which did not make a matching set of "#" symbols so FLAG2# was simply passed through also.

All of the above may seem a bit far fetched but the features explained are very useful when one wants to use the CHAIN command from within a CHAIN procedure. For example, one could pass the date from one procedure to the next by having the procedure file:

```
ASM TEST;XL
TEST PROGRAM ASSEMBLED ON #DATE#
CHAIN CHAINF2;DATE##DATE##
```

Note that if a tag is mentioned in the CHAIN command line but given no value and if the value is called for substitution, a null value will be substituted for the #<tag># within the line. The effect is that the #<tag># characters simply disappear from the line. For example, a CHAIN command:

```
CHAIN MAKETEST;DATE
```

made for the procedural file shown above (assume it was called MAKETEST) would result in a CHAIN/SYS file containing:

```
ASM TEST;XL
TEST PROGRAM ASSEMBLED ON
CHAIN CHAINF2;DATE##
```

14.3.6 Additional CHAIN operators

In addition to the operators mentioned in previous sections, CHAIN contains ABORT, BEGIN, and END operators. The ABORT operator simply causes CHAIN to return instantly to DOS without any further action. If any messages are to be given to the operator, it is the responsibility of the procedural file to contain the appropriate compile time comments before the ABORT operator. This operator makes it easy to terminate the procedure if a critical tag is missing or some other problem with the

procedural file or its parameterization is detected.

The BEGIN and END operators allow groups of IF/ELSE/XIF operators to be parenthesized. A counter called the BEGIN/END counter is initialized to zero when compilation of a procedure begins. If the use of procedural lines is turned off and a BEGIN operator is encountered, then the BEGIN/END counter is incremented. If an END operator is encountered, then the BEGIN/END counter is decremented unless it is already zero. The ELSE and XIF operators have no effect if the BEGIN/END counter is not equal to zero. For example:

```
// IFS FLAG1
ASM TEST1;XL
TEST PROGRAM ONE
// ELSE
// BEGIN
// IFS FLAG2
ASM TEST2;XL
TEST PROGRAM TWO
// ELSE
ASM TESTTEST;XL
TEST TESTER
// XIF
// END
// XIF
// IFS FLAG3.FLAG27
LIST SCRATCH;L
THE SCRATCH FILE AT FLAG 27
// XIF
```

The 6th through the 12th lines will not be used if FLAG1 exists, notwithstanding the fact that there is an ELSE and XIF operator within those lines, because the BEGIN/END pair prevented these operators from having any effect.

14.3.7 Resuming an aborted CHAIN

Before the CHAIN overlay fetches the next DOS command it stores the CHAIN/SYS file pointers for the line to be used. If something goes wrong during the DOS command which follows and the procedure is aborted, CHAIN still knows where it was in the CHAIN/SYS file when the problem occurred. Since CHAIN does not delete the CHAIN/SYS file unless the procedure completes successfully, it can pick up where it stopped in the CHAIN/SYS file if the operator can correct the condition which caused the procedure to abort in the first place. Often, the reason for the abort is something correctable like the disk running out of files

or an attempt to delete a non-existent file. In this case, the operator need only correct the condition and then enter:

CHAIN *

and the procedure will pick up with the command which failed before. This action can generally be applied even if the RESTART key has been depressed. Thus, one can recover from jammed paper in a printer half way through a listing by simply depressing RESTART, fixing the printer, and then entering the CHAIN * command.

If the failing command cannot ever succeed, it may be bypassed by entering the command:

CHAIN/OV1

This simply restarts the chain with the next available line in the procedure. If the next line had been intended as a keyin line for the failed program (as opposed to a DOS command line) the chain will generally immediately abort again. However, by restarting the chain in this manner, repeatedly if necessary, the invalid step can usually be bypassed and chaining resumed.

CHAPTER 15. CHANGE COMMAND

CHANGE - Change a file's protection

CHANGE <file spec>;p

The CHANGE command enables one to write protect, delete protect, or clear the protection of a disk file. If a file is delete or write protected, a KILL command (or program generated kill) cannot affect it. If a file is write protected, it cannot be written into by the standard system routines.

The option parameter "p" is used above to indicate the protection for the file specified. Protection can be specified as:

D - delete protect
W - write protect
X - clear protection.

For example:

```
CHANGE NAME/EXTENSION;D
CHANGE NAME/EXTENSION:DR2;X
```

will delete protect the file in the first case, and remove all protection in the second case. If a first specification is not given, the message

NAME REQUIRED.

will be displayed. If the file indicated by the first file specification cannot be found, the message

NO SUCH NAME.

will be displayed. If the option parameter does not follow the above syntax rules, the message

INVALID PROTECTION SPECIFICATION.

will be displayed.

CHAPTER 16. COPY COMMAND

16.1 Purpose

It is frequently useful to make a copy of a disk file. It may be desired, for example, to make a copy on a separate volume for backup or distribution purposes.

It is possible to copy a disk file using more specialized commands such as SAPP and APP. However, since these commands make assumptions regarding the internal details of the contents of the file they are copying, they require one to distinguish between standard DOS TEXT-type files and OBJECT files (files whose contents are in the standard format acceptable to the DOS loader). Additionally, SAPP and APP cannot correctly copy certain unusual types of files, for example those with imbedded end-of-file records, physical random data files, and the like.

Because the COPY command does not make assumptions about the format of the sectors being copied, but merely copies the file sector-for-sector, it can copy most types of disk files which previously were not possible to copy using the SAPP and APP commands. Some particular types of file are still unmovable, however. The outstanding example are INDEX files, usually with extension /ISI. (These cannot be moved because index files contain, internal to themselves, pointers indicating their actual physical location on the disk volume, which are made invalid when the file is moved to another place on the disk).

Another advantage of the COPY command is that since sectors are not examined for content, some versions of the command can copy files much faster than is possible using APP or SAPP.

16.2 Use

The COPY command is invoked by entering at the system console:

COPY <input file spec>,<output file spec>

The COPY command causes the first specified file to be copied into the second one. Attributes of the first file, such as its protection, are copied to the second file as well.

The only portion of the operands that is specifically required is the name of the input file. The extension of the input file, if none is specified, is assumed to be /TXT. If a drive specification is entered for the input file, then only that specific drive is searched for the indicated file. If no drive specification for the input file is given, all drives are searched. If the name of the output file is omitted, it is assumed to be the same as that of the input file. If the output file's extension is not given, it is also assumed to be the same as that of the input file. All drives are searched for the output file before creating it unless a particular drive is specified.

For example, to copy file PAYROLL/TXT from drive two to drive one, it is only necessary to enter:

```
COPY PAYROLL:DR2,:DR1
```

As another example, to make another copy of PROGRAM/ABS on drive zero, but to be named MYPROG, all that is required is:

```
COPY PROGRAM/ABS,MYPROG:DR0
```

People who experience parity errors in one of their data files can frequently recover their data using COPY. Since the COPY program merely comments about parity errors encountered and does not abort when one occurs, the data copied will occasionally be correct (or almost correct) even if a parity error occurs and can be used to recover the data in the original file. Alternatively, using the COPY program to write the file on top of itself (therefore without changing the file) by simply specifying the input file and no output file, a user can frequently clear soft (and occasionally what seem to be hard) parity errors occurring in an important data file. (Of course, no important file should be updated in place unless a copy of the file exists somewhere for recovery purposes in the event of a failure).

Some versions of the COPY command issue a click each time an unwritten sector is copied. If more than a dozen or so clicks occur at the end of copying a file, it usually indicates that the file is larger than necessary to contain the data in it. In this case, moving the file using APP or SAPP can sometimes help to reduce its size. Clicks occurring during the copying (before the end of the file) indicate sectors containing DOS format errors, possibly implying a sector accidentally destroyed by some faulty program.

CHAPTER 17. DOSGEN COMMAND

17.1 Purpose

Before any disk can be used by DOS, certain tables and other information must be placed onto it to establish the basis that DOS requires for the support of its file structure. These tables include the skeleton of the DOS directory, where the names of the files contained on the disk are stored, as well as a map showing which places on the disk are bad and should not be used.

The purpose of the DOSGEN command is to provide the user with a simple and efficient way of accomplishing this.

17.2 Use

To DOSGEN a disk, simply enters:

DOSGEN <drive spec>

The drive spec is a standard DOS drive specification which specifies which drive contains the disk to be prepared for DOS use. Since the directory initialization process will effectively KILL any files that might be on the disk, the command asks several times to make sure that the operator is aware of the potential seriousness of the operation he has invoked.

After the operator has acknowledged that he does not mind the overwriting of the new disk, the command asks if any cylinders on the volume are to be locked out. Normally, the answer to this question is NO. However, by answering YES, it is possible to cause the DOS to lock out one or more cylinders of the disk from DOS access. This can be useful in some special applications where it is desired to not allow DOS programs access to a file stored in unusual format, for example. In general, locking out cylinders from DOS access is to be discouraged since it makes it more difficult to make use of the useful features of the DOS. If the user does wish to lock out any cylinders, he may do so by specifying one or more cylinder numbers, in the format:

12,14,16,25-28,40

The above example would cause cylinders 12, 14, 16, 25, 26,

27, 28, and 40 to be locked out. Note that the cylinder numbers to be locked out are given in decimal as opposed to octal.

After the operator has specified that no, or which, cylinders are to be locked out, the DOSGEN command checks for bad sectors on the disk and issues a message indicating any cylinders it finds which contain bad sectors. The remainder of the operation is completely automatic and indicates its completion with the familiar DOS message, "READY".

Upon completion of the DOS generation process, the only files on the new disk are the eight system files SYSTEM0/SYS through SYSTEM7/SYS and the CAT command.

17.3 Special Considerations

It is important to remember that on disk packs for use with DOS systems recognizing more than one logical drive per physical disk pack, for example the 9370 series disk system, two DOSGENs must be done before the physical pack is fully initialized. This allows the user to DOSGEN either logical disk on the pack without disturbing files he wishes to keep that may be stored on the other logical disk.

Another important thing to remember is that both the 9370 and 9380 series disks must be <u>formatted</u> before DOSGEN can be used on them. Diskettes (for the 9380 series drives) come pre-formatted from the manufacturer; disk packs for the 9370 series drives do not. It is therefore necessary to format all disk packs for the 9370-series drives using the program INIT9370 before attempting to use DOSGEN on them. A diskette that has been formatted with tracks locked out (error mapped) cannot be DOSGENed.

CHAPTER 18. DUMP COMMAND

18.1 Purpose

Occasionally while writing into files on disk (in particular, during the program debugging stage) it is useful to be able to verify that the formatting of the information into the standard text format is being done correctly. Or, perhaps an assembler language program (/ABS file) that previously loaded correctly no longer will, as indicated by DOS just coming back up when the program is run.

The DUMP command provides a simplified mechanism for examining the entire contents of physical sectors on the disk. The display includes both the octal and ASCII contents of every byte on the sector. No examination for control bytes of any kind is made, allowing the user to see the precise contents of every physical location in the disk sector.

Another good use for the DUMP command is to clarify any questions regarding the standard DOS file formats. Using DUMP it is possible to examine a file and see just how it is formatted on the disk. DUMP is frequently useful for DATASHARE programmers who are using tabbed reads and writes and encounter problems with their programs, since these problems are usually caused by a lack of complete understanding of the format of DOS standard text files and how this interacts with tabbed disk operations in DATASHARE.

18.2 Use

The DUMP command is invoked by entering:

```
DUMP
```

or

```
DUMP <file spec>
```

The DUMP command operates with basically five separate levels of control. These levels are:

LEVEL ONE - Logical drive level
LEVEL TWO - File level

LEVEL THREE - Logical record number level
LEVEL FOUR - Physical disk address level
LEVEL FIVE - Disk directing level

The (optional) entry file and/or drive specifications on the command line allow the first one or two input levels in DUMP to be automatically bypassed.

When the DUMP command is used, the top line on the display is the primary control line. Input typically is accepted on this line. This line is broken into four basic areas, one corresponding with each of the first four control levels. The primary control level at any given time during the operation of the DUMP command can be determined by the position of the flashing cursor on the control line.

For example, if the flashing cursor is positioned after the "DRIVE:" legend on the control line, the DUMP command is operating at level one. If the cursor is positioned after the "FILE:" legend on the control line, the DUMP command is operating at level two, etc.

18.3 Informational Messages Provided

The second line on the display is primarily used for sector informational messages. These serve both to indicate any special significance of the sector just read and to describe any unusual occurrences associated with reading the sector. These messages are generally self-explanatory. Among the messages that can be displayed are the following, along with an explanation of the meaning of each.

RETRIEVAL INFORMATION BLOCK (RIB). This message indicates that the sector being displayed is the primary RIB for the currently opened file.

RETRIEVAL INFORMATION BLOCK BACKUP. Each RIB is maintained in duplicate for backup purposes and to allow recovery in the event of a program erroneously destroying the primary RIB. This message indicates that the sector being displayed is the secondary RIB for the currently opened file. Note that this does not mean that the primary RIB has necessarily been damaged; it simply means that the sector requested happens to be the secondary, backup copy of the RIB.

CLUSTER ALLOCATION TABLE. This message indicates that the sector being displayed is the primary Cluster Allocation Table (normally referred to as the CAT) for the current logical drive.

CLUSTER ALLOCATION TABLE BACKUP. This message indicates that the sector being displayed is the secondary, backup CAT for the current logical drive. This implies that the CAT is also maintained in duplicate just as is the RIB.

LOCKOUT CLUSTER ALLOCATION TABLE. Associated with each logical drive is a sector that indicates which areas have been locked out, prohibiting their use by DOS. This message indicates that the sector being displayed is the Lockout CAT for the current logical drive.

LOCKOUT CLUSTER ALLOCATION TABLE BACKUP. This message indicates that the sector being displayed is the secondary, backup copy of the sector just described above.

SYSTEM DIRECTORY SECTOR. This message indicates that the sector being displayed is one of the DOS directory sectors. The directory sector number (in decimal and in octal) immediately follows the message.

USER DATA SECTOR. This message indicates that the sector is not recognized as one of the above special system sectors.

DISK SECTOR CRCC ERROR. This message indicates that the sector requested for display either was not found on the disk or that a CRCC error repeatedly occurred during the read operation. The sector displayed is the data as it was read from the disk, unless the sector was not found.

DISK OFFLINE. This message indicates that the currently specified logical drive is not on line.

DISK SECTOR FORMAT ERROR. This message is displayed when DUMP notices that the sector being displayed does not correspond to standard DOS file conventions (the first byte of each sector is its physical file number, and the two following bytes are the logical record number). The appearance of this message does not necessarily indicate that the sector of the file has been destroyed, since unwritten sectors at the end of a file and older version DATASHARE object code files normally will fall into this class. It merely means that if the sector were read with the DOS READ$ routine, a format trap would occur.

SECTOR OUT OF RANGE. This message is displayed if the sector requested (by logical record number) is not within the range of the currently opened file.

FILE NOT FOUND. This message indicates that the file

requested could not be found. This does not necessarily mean that the file does not exist. For example, the file could be in a non-current subdirectory. If the user has not requested non-specific volume mode (to be described), this message might mean simply that the file desired is on a different logical drive.

INVALID PHYSICAL ADDRESS. This message indicates that the physical disk address specified is invalid.

The remainder of the display contains the contents of the current half of the sector most recently read. The display is arranged as eight groups of sixteen bytes each. Each of these groups is preceded by the three octal digit offset of that group within the sector. Each sixteen byte group consists of the octal and ASCII contents of each of the sixteen bytes in that group. Each byte's contents form a column one byte wide and four lines high, where the first three lines are the value of the byte, in octal, and the fourth line is the ASCII value of that character. Notice that the character is not examined for special significance before it is displayed, so that computers having the high speed RAM display option (which is strongly recommended for all DOS systems) may display characters other than the normal ASCII set.

18.4 Level One Commands To DUMP

When the flashing cursor indicates that DUMP is functioning at level one, the following commands are accepted:

<enter> - The CAT on the current drive is displayed and control is transferred to level two. In addition, the non-specific drive mode is enabled.

number - The drive number indicated becomes the currently selected drive. The CAT from that drive is displayed and control is transferred to level two. Non-specific drive mode is disabled.

* - DUMP command returns control to the DOS.

> - The second half of the current sector is displayed.

< - The first half of the current sector is displayed.

18.5 Level Two Commands To DUMP

When the flashing cursor indicates that the DUMP command is functioning at control level two, the following commands are accepted:

<enter> - If a file is currently opened, the secondary RIB for the file is displayed and control is transferred to level three. If no file is opened, control is transferred to level four.

name/ext - The named file is opened on the current drive, or any drive if non-specific drive mode is enabled. The primary RIB for the file is displayed and control is transferred to level three.

pfn - The file indicated by the octal physical file number given is opened on the current drive. The primary RIB for the file is displayed and control transfers to level three.

I - The current physical file number is incremented and the new file thus indicated is opened. If no file corresponding to that physical file number exists on the current drive, the PFN is incremented repeatedly until a file corresponding to the PFN is found. The primary RIB for the file is displayed and control is transferred to level three.

D - D works just like the I command above except that instead of incrementing the PFN, it is decremented.

#pfn - The directory sector containing the entry corresponding to the file indicated by the specified physical file number is displayed; then control is transferred to level five. Since only the last four bits of the PFN are relevant, the pfn specifier is equivalent to a relative directory sector number. These directory sector numbers are always specified in octal.

* - Return control to level one.

> - Show the second half of the current sector.

< - Show the first half of the current sector.

18.6 Level Three Commands To DUMP

When the cursor indicates that DUMP is functioning at level three, the LRN level, the following commands are accepted.

<enter> - The current sector is shown and control is transferred to level four.

number - Access and display the record indicated by the LRN specified. If the number given has a leading zero, it is assumed to be octal; otherwise it is assumed to be decimal. The number specified is the user (as opposed to system) LRN. The system LRN, the one in bytes one and two in the sector, is always two less than the user LRN. The two numbers displayed at level three in the control line are the LRN in decimal (the one with leading zeros suppressed) and octal (the one in parentheses, with leading zeros).

I - Increment the current logical record number, access it and display the sector.

D - Decrement the current logical record number, access it and display the sector.

* - Return to the File level of control (level two).

> - Show the second half of the current sector.

< - Show the first half of the current sector.

18.7 Level Four Commands To DUMP

Level four of the DUMP command requires more detailed understanding of DOS physical disk addresses, and as such is not usually as useful as the LRN level. However, when access to a specific sector on the disk is desired, it can be achieved using DUMP level four. It is important to realize that the physical disk addresses specified are logical physical disk addresses, i.e. the same format as is given to the DR$ and DW$ routines in the DOS. They are not necessarily the same as actual physical locations on the disk. For example, with DOS.C for the 9380 series diskettes, the logical disk addresses are remapped onto the diskette into different hard physical sector numbers than those indicated by the logical physical disk address. The important thing to understand here is that the disk addresses used in the level four control of DUMP are those that would be used to parameterize DR$ and DW$.

The commands accepted at level four of DUMP are as follows.

msb,lsb - Access and display the sector indicated at the given physical disk address on the current logical drive. The first field (most significant byte) is assumed to be in decimal unless a leading zero is supplied. The second field (least significant byte) is always considered to be in octal, regardless of whether a leading zero is supplied or not. The second field is separated from the first by a comma. The physical disk address given by the user is assumed to be valid. If it is not of the proper format, undefined results may occur. Users who are not sure of their understanding of DOS internal physical disk addresses should not use level four of DUMP.

* - Return control to level two if no file is opened, or level three otherwise.

> - Show the second half of the current sector.

< - Show the first half of the current sector.

18.8 Level Five Commands to DUMP

When the flashing cursor indicates that the DUMP command is operating at control level five (system directory sector level), the following commands are accepted:

number - Show the directory sector indicated by the low order four bits of the number specified. Since only the low order four bits of the number are used, it is not an error to specify simply the physical file number (PFN) of the file whose directory entry is to be examined. A leading zero indicates the number is in

octal, otherwise decimal is assumed.

I - The current directory sector number is incremented and the corresponding directory sector is displayed.

D - The current directory sector number is decremented and the corresponding directory sector is displayed.

* - Return control to level two.

> - Show the second half of the current directory sector.

< - Show the first half of the current directory sector.

18.9 Error Messages

Only one genuine error message is issued by the DUMP command. It is:

ERROR IN DOS FUNCTION. DUMP ABORTED.

If this error message occurs, it means that the DOS FUNCTIONs are probably incorrect on the disk, generally indicating that the disk in drive zero has not been completely (or correctly) DOSGENed. If this is the case, SYSTEM7/SYS should be loaded using the latest copy of DOS as distributed by Datapoint.

CHAPTER 19. EDIT COMMAND

19.1 Introduction

The DOS Editor is used to create and to update source data files on the disk. The editor, through the use of initialization parameters, will enable the creation of files in a variety of formats: text files, assembler code files, DATABUS source code files, or many user designed data files.

A GLOSSARY of the many terms and phrases used throughout this chapter is provided in the Glossary at the end of the chapter. A list of commands and brief definitions is provided in the COMMAND List Section. Caution: Although virtually any Datapoint format file may be "edited", files structured with respect to physical records or those containing strings longer than 79 characters may have this organization collapsed as the editor compresses the file into sequential format. In such cases the editor should not be used.

The editor does not trancate trailing blanks at the end of lines unless it is in "COMMENT" mode.

19.2 Operation

19.2.1 DOS Initialization

The EDIT program, is parameterized as follows:

EDIT <f1>[,<f2>][,<f3>][;parameter list]

19.2.2 Files

<f1> is the source file, [<f2>] is the scratch file and [<f3>] is the configuration overlay file. The source file <f1> is assumed to have an extension of 'TXT' if none is provided. If there is no file of the specified name, one will be opened. If no scratch file [<f2>] is specified, a file 'SCRATCH/TXT' will be used. The configuration file [<f3>] is assumed to be EDIT/OV1 unless otherwise specified. WARNING: The default extension for the configuration file is 'OV1'.

If parameters are indicated by the presence of the semi-colon, the question:

RECORD PARAMETERS?

will be displayed. If 'N' is entered, the editor will begin execution with the indicated parameters and the configuration file will not be changed. If 'Y' is entered, the question:

NEW TABS?

will be asked. If 'Y' is entered, the standard tab initialization line of numbers will be displayed (see :T command description). After the new tabs are entered, the parameter information and tabstops are recorded in [<f3>].

If no parameter list is provided, [<f3>], if present, is automatically loaded, causing the recorded parameters to be used.

19.2.3 Parameter List

A parameter list, indicated by the SEMI-COLON (;) following the file specifications may be included. That list may include up to seven parameters which are order independent. The possible parameters are:

[;[margin][tab key][mode][shift][line][update]

If no parameter list is provided, Assembler mode with a margin at 75 and SPACE bar for tabbing is assumed.

19.2.3.1 Margin Bell

A number in the parameter list will be taken to be the margin designator; this causes the margin 'bell' to ring at the designated margin. (Text may always be input up to column 79 regardless of the margin setting.)

For Example ;30 will cause the bell to ring in column 30.

19.2.3.2 Tab Key Character

A tab key character encountered in the parameter list, i.e., a non-alpha, non-numeric, non-colon, will replace the assumed tab key character. (SPACE in Assembler, DATABUS and Comment mode, SEMI-COLON in Text mode.)

For Example, ^ will cause the caret key (^) to replace the

assumed character as the tab key.

19.2.3.3 Mode

A new set of assumptions will be used if one of the 'mode' parameters is set. If no mode is listed or 'A' is typed, Assembler mode will be used. DATABUS or DATAFORM (D) mode simply changes the tab stops. Comment mode (C) changes the nature of the DELETE and SCRATCH commands to facilitate adding or changing comments on assembly code files and also truncates trailing spaces.

Text mode (T) sets no tabstops, does no shift inversion and enables the word wrap around feature (see the glossary). To activate line truncation instead of word wrap around in Text mode, enter 'L' in the parameter list. To enable shift key inversion (see glossary) in Text mode, enter the parameter 'S' in the list. Text mode is especially useful for generating SCRIBE input files.

See the glossary for complete definitions of the various modes.

19.2.3.4 Update

During editing, the source file is transferred into the scratch file as the text is updated. The physical source file may be used as the scratch file as the edit proceeds. When the edit is terminated, the physical source file is updated.

To inhibit source file update, the 'ONE-PASS' parameter 'O' may be set in the parameter list. A flag is set which prevents writing on the physical source file. Then, at the completion of the edit, the scratch file will contain the updated information and the source file will be unchanged.

19.2.3.5 Key-click

If the 'K' parameter is set, a click will sound each time a key is struck.

19.2.4 Examples

To perform standard Assembler code editing, enter the command:

EDIT <source>

To edit a file for input to the text processor, SCRIBE, enter the

command:

EDIT <source>;T

To change the margin bell to ring at column 35 (e.g. for labels) enter the command:

EDIT <source>;35T

The parameters would set the bell and use the Text mode assumptions. Note that the parameters are order independent; therefore, the command:

EDIT <source>;T35

would achieve the same results.

To generate a second, slightly different, file (without updating the original file), enter the command:

EDIT <source>,<new file>;OT

If the file is Assembler code instead of text, simply omit the 'T'; if DATABUS, replace 'T' by 'D'.

A second file, with the same name as <f1> but with a different extension, may be used as the scratch file by entering:

EDIT <f1>,/<extension>

Once the initial command (and parameter list) has been entered, the DOS Editor signon message will appear on the screen. This message will be rolled up and the screen cleared with the cursor left on the 'command line'. From this position data may be entered, lines may be fetched from the source file, or editor commands may be entered.

19.2.5 Data Entry

To enter text, simply type on the bottom line; when the ENTER key is pressed the screen rolls up one line. The command line is once again blank and the cursor is at the beginning of the command line, ready to accept more input.

If word wrap around is enabled, when a SPACE is typed within the last 10 columns of the line or typing proceeds past the end of the line, the editor automically will roll up the screen and begin a new line. If a non-space character is typed into the last

column, the last word on the line is removed and, after the screen is rolled up, that word is place on the command line, where data entry may proceed.

When typing on a 'screen line' (as the result of a command), the ENTER key causes the cursor to return to the command line. To continue data entry at the same screen area, the Pseudo-ENTER key may be used. This key (DEL shifted) causes (in all but command mode), a new blank line to be inserted at that point on the screen so that data entry may proceed.

If word wrap around is enabled, and data is being entered on a screen line, a new line will automatically be inserted at that point when, as on the bottom line, a space is entered within the last 10 columns or a character is typed past column 79.

The BACKSPACE key erases the last character and moves the cursor back one position. The CANCEL key erases the line back to the previous tabstop (in text mode this would erase the entire line if no tabs are set).

Typing the tab key character causes the cursor to move to the next tab stop to the right. If there are no tab stops to the right of the cursor, the tab key character is accepted as a normal data character.

19.2.6 Data Retrieval

To fetch data from the source file, press the KEYBOARD and DISPLAY keys simultaneously. As long as the two keys are depressed, data will be fetched, displayed on the command line and rolled up the screen. If end of file is reached, no more data is fetched and the machine beeps.

To fetch a single line, the shifted DEL key may be pressed (in the first column of the command line). Using this key insures that only one input line will be fetched.

19.2.7 EDITOR Command Format

The text appearing on the eleven screen lines (i.e. the lines above the command line) may be edited using a set of 'commands'. A 'pointer' (>) in the left hand column of the screen indicates the line which the command will affect.

To move the pointer up, press the KEYBOARD key. To move the pointer down, press the DISPLAY key. The pointer wraps around from the top to the bottom and vice versa.

Commands allow the user to delete a single line (:D) or part of the screen (:SC and :SB), insert (:I) a new line between the current lines on the screen and modify (:M) parts of a line by replacing text or inserting new text. Commands are also available to search the file for specific text (:F and :L) or for the end of the file (:EO or :E*).

An editor command is always preceeded by a COLON (:). To enter a command, type, in the first column of the command line, a colon and the appropriate command character and any necessary parameters. The command is always typed with the machine in lower case; thus, with shift inversion on (as in Assembler, Databus and Comment modes), the command character will appear upper case; while with shift inversion off (as in Text mode), it will appear lower case.

19.3 Basic EDITOR Commands

The following commands are a few basic editor commands. The user can get started without worrying about complex command forms. Remember that the 'pointer' on the screen indicates the line affected by the command.

:D - DELETE - in all but Comment mode this command deletes the entire pointed line. (In Comment mode, only the comment field is deleted. The CANCEL key may however be used to delete the preceeding fields in the line.)

The cursor is left on the now null line where new text may be entered. If no replacement text is needed, pressing the ENTER key in the first column of the pointed line returns the cursor to the command line. Trailing blanks will not generally be truncated.

Pseudo-ENTER may be used to generate additional lines at this area of the screen. Word wrap around, if enabled, will apply to text entered on a deleted line. Pressing the ENTER key will return the cursor to the command line.

See the section on modification for more information about the pseudo-ENTER key.

:E* - EOF without display - searches for the end of the file and, when it is reached, displays the last eleven lines of text. The search may be aborted by pressing the KEYBOARD and DISPLAY keys simultaneously.

:EO - EOF with display - causes the data to be displayed on the screen continuously until end of file is reached. The search

may be aborted at any time by pressing the KEYBOARD and DISPLAY keys simultaneously.

:F <old text> - FIND match - the screen is cleared and the input file is searched for a line starting with the specified <old text>. Leading spaces in the file's lines will be ignored and should not be entered as part of <old text> (note that this command should be typed exactly :F<SPACE><old text>).

A FIND will wrap entirely around the file (or up to the end of file if the one-pass option is set). If the requested text is not found, the last line on the screen when the FIND was executed will be displayed. A FIND may be aborted by pressing the KEYBOARD and DISPLAY keys simultaneously.

:I - INSERT - Perform a line insert at the pointed line. This command causes the lines from the top of the screen to the pointed line, inclusive, to be rolled up and a blank line to be inserted. The cursor is left at the beginning of the new blank line where data entry may proceed.

If the pointed line or the line immediately below it is empty no insert will occur, and the null line will be used as the inserted line where data entry may proceed.

To make complex changes to a line already on the screen, the operator may INSERT a line immediately below the original and then retype the line - with changes. The original line may then be DELETED.

The pseudo-ENTER key may be used to generate additional lines at the same point on the screen.

:L - LOCATE next - typed exactly :L<ENTER>, clears the screen and finds the next line of text. If positioned at the end of the file, the 'next' line will be the first line of the file.

:L <old text> - LOCATE match - similar to FIND match except that the locate command searches for imbedded text matching <old text>. Leading spaces should be supplied if meaningful.

For additional forms of the FIND and LOCATE commands see the 'FILE SEARCH' section.

:M <old text><command separator><new text> - MODIFY - a modify command allows the operator to replace <old text> by <new text>, insert <new text> after <old text> or append (i.e., truncate and add) <new text> after <old text>. For the various

forms of this command see the 'MODIFICATION' section.

:SC - SCRATCH above - in all but Comment mode this command erases the lines from the top of the screen down to the pointed line, inclusive. (In Comment mode, only the comment fields are erased.)

The cursor is left on the pointed line where data entry may proceed.

:SB - SCRATCH below - in all but Comment mode this command erases the lines from the pointed line to the bottom of the screen, inclusive. (In Comment mode, only the comment fields are erased.)

The cursor is left on the pointed line, where data entry may proceed.

:E - END - the end command causes the remainder of the logical source file to be copied to the logical scratch file and then, if the logical scratch is not the physical input file, the scratch file is copied back to the source file.

The command line will be left on the screen as long as the copy from source to scratch is in progress; it is erased during the final copy from scratch back to source.

The end may be aborted as long as the command line is still displayed, by pressing the KEYBOARD and DISPLAY keys simultaneously. When the final copy is completed, control is returned to DOS.

Note that if the one-pass option was selected in the parameter list, no copy from scratch back to source will be performed.

:E/ - END/DEL - this command causes the remainder of the source file to be deleted (the lines currently on the screen will be written out), and, if the logical scratch file is not the physical source file, the scratch file is copied back to the source file. When the file is completely updated, the system is reloaded.

No copy back is done if the one-pass option is set.

19.4 Modification Commands

19.4.1 DELETE Command

Modification of a line may be achieved in a variety of ways. The DELETE command enables the user to remove leading information while the MODIFY command may be used to replace imbedded information, insert text into a line or field, or truncate and add new text at a specified point or in a specified field.

:D <old text> - DELETE through - this command deletes all character from the left edge of the pointed line through (and including) the specified <old text>. The remaining characters will be left justified and re-displayed. The cursor returns automatically to the command line.

19.4.2 MODIFY Command

The general form of the MODIFY command is:

:M[#] [old text]<sep>[new text]

where [#] is an optional number which extends the meaning of the command (see FIELD MODIFICATION below) and <sep> is the command separator which defines the action of the command. Both [old text] and [new text] fields are optional. If [old text] is omitted, the command will take effect at the left most edge of the pointed line (or at the left edge of the specified field). If the [new text] field is omitted, a null field will be used to execute the modification.

19.4.2.1 Line Modification

The following descriptions are of the line modification version of the MODIFY command

:M [old text] < [new text] - MODIFY (replace) - replace the specified [old text] by the specified [new text]. The less than character (<) is a command separator which indicates replacement and, therefore, the [old text] may not contain this character. If [new text] field is omitted, the old text will simply be deleted and the line will be compressed to the left.

For example to modify the text line:

THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG'S BACK.

The command: :M BROWN<RED would cause the line to be redisplayed like this:

THE QUICK RED FOX JUMPED OVER THE LAZY DOG'S BACK.

The command: :M .< 1234 TIMES. to the original line would generate a line like:

THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG'S BACK 1234 TIMES.

If the replacement causes the line to become longer than 79 characters, the trailing word, in text mode only, will be wrapped around and a new line will be inserted containing the entire last word. If the [new text] is shorter than the [old text] it replaces, the line will be shortened.

After the pointed line is redisplayed, the cursor is returned to the command line.

:M [old text] > [new text] - MODIFY (insert) - the command separator greater than (>) causes the [new text] to be inserted in the pointed line immediately after the [old text].

If the line becomes longer than 79 character, and word wrap around is not in effect, the trailing characters are truncated. If, however, word wrap around is on, the trailing character and last word are inserted on a new line.

:M [old text] \ [new text] or

:M [old text] | [new text] - MODIFY (append) - the vertical bar (|) or backslash (\) command separators cause everything in the pointed line, past the [old text], to be replaced by the [new text].

As in all MODIFY commands, if the pointed line becomes longer than 79 characters, truncation occurs if word wrap around is not enabled.

:M[#] - MODIFY repeat - typed exactly :M<ENTER>, uses the <old text> <sep> <new text> from the last MODIFY command. This is useful when making the same change repeatedly. Note that the field number is not saved, and must, therefore, be supplied if necessary.

:M* - MODIFY display - display the expression entered for the last MODIFY. After the saved command is displayed, the cursor is turned off and the operator must press ENTER to proceed. No MODIFY is actually performed.

19.4.2.2 Field Modification

In field modification mode, the MODIFY command acts only on a specific field and does not expand or contract the entire line but maintains the integrity of all fields before and after the affected field.

A field is the area between two consecutive tabs. Field one is between the left margin and the first tab.

:M<#> [old text]<sep>[new text] - MODIFY field - where the pound sign <#> is a number from 1 to 10 designating the field to be modified (or the starting point to search for matching [old text]). In Assembler mode, field 1 is the label, field 2 is the op code, field 3 is the expression and field 4 is the comment. This command may be executed in any of the previous Modify forms. However, modification is performed within the specified field only. As long as the text being modified is unique, field 1 may be specified, since the field number indicates only where to start looking for matching text. (Note that if the field number is omitted, line modification is assumed.)

Thus, a replacement or append shorter than the original field will be blank filled and subsequent fields will maintain their position and content. An insertion longer than the specified field will be truncated (with the exception of the last field whenever word wrap around is in effect).

For example, in Assembler mode, the line:

```
LABEL     OP       EXP              COMMENT
```

the label may be deleted by the command:

```
:M1 \
```

with the resultant line:

```
          OP       EXP              COMMENT
```

Or, the expression field (EXP) could be changed to EXP+1 without disturbing the comment field position, by the command:

:M3 EXP>+1

which generates:

LABEL OP EXP+1 COMMENT

To add a comment to a line previously containing none or to replace an existing comment field, enter:

:M4 \<new comment>

NOTE: Remember when using the repeat form of the MODIFY command that the field number may need to be supplied.

19.5 File Search Commands

The FIND and LOCATE commands have several forms and have been separated from the basic command set to better describe them.

Manual, operator controlled, searches may be performed by depressing the KEYBOARD and DISPLAY keys simultaneously to cause data to be fetched from the file and displayed (as long as the keys are pressed) on the screen. To fetch a single line use the Pseudo-ENTER key (DEL shifted). The :EO command performs the same function automatically, i.e., it causes lines to be fetched and displayed until the end of file is reached. To abort a :EO command, press the KEYBOARD and DISPLAY keys simultaneously.

To find the end of a file without displaying the entire file (since the display is time consuming) use the :E* command. This will search for the end of file and display the last eleven lines of data.

:F <old text> - FIND match - the screen is cleared and the input file is searched for a line starting with the specified <old text>. Leading spaces in the file's lines will be ignored and should not be entered as part of <old text> (note that this command should be typed exactly :F<SPACE><old text>).

A FIND will wrap entirely around the file (or up to the end of file if the one-pass option is set). If the requested text is not found, the last line on the screen when the FIND was executed will be displayed. A FIND may be aborted by pressing the KEYBOARD and DISPLAY keys simultaneously.

The <old text> specified for a FIND (or LOCATE) command is saved. The saved match may be redisplayed or used again.

:F<SPACE> - FIND same match - if the FIND command is followed by exactly one space and the ENTER key, the previous FIND (or LOCATE) <old text> will be used for this FIND. Several occurrences of the same text may be searched out in this manner.

:F* - FIND display - the asterisk (*) after the FIND command causes the <old text> of the previous FIND or LOCATE command to be displayed. The cursor is turned off and the operator must press ENTER to proceed. No FIND is performed.

:L - LOCATE next - typed exactly :L<ENTER>, clears the screen and finds the next line of text. If positioned at the end of the file, the 'next' line will be the first line of the file.

:L <old text> - LOCATE match - similar to FIND match except that the locate command searches for imbedded text matching <old text>. Leading spaces should be supplied if meaningful.

:L<space> - LOCATE same match - typed exactly :L<SPACE><ENTER>, uses the <old text> specified by either the previous LOCATE or FIND command to perform a search.

:L* - LOCATE display - display the <old text> entered for the previous LOCATE or FIND command. As in the FIND display, the cursor is turned off and the operator must press ENTER to continue. No LOCATE is actually performed.

19.6 Miscellaneous Commands

:A - APPEND - copies the pointed line to the bottom of the screen and rolls the screen up one line.

:B - BYPASS - fetch a line from the file, bypassing end of file or record format error (which would normally be treated as an end of file). Subsequent lines (if not also record format errors) may then be fetched by the normal mechanisms. This command is intended as a recovery tool for use only if the file has been accidentally shortened or contains badly formatted records.

:C - COPY - copies the pointed line to the bottom of the screen, deletes the pointed line and rolls the screen up one line. This command cannot be executed on the top screen line.

The cursor is left on the now null pointed line. Text may be entered at this point (the Pseudo-ENTER and word wrap around, if enabled, will apply). When the ENTER key is finally pressed, the pointer is automatically moved to the following screen line so

that a group of lines may be easily copied to another part of the screen.

:T - TAB set - this command enables the user to reset the tab stops during execution. (Not available in Comment mode.) The command causes a line of numbers to be displayed across the bottom of the screen.

The operator should space over to each position where a tabstop is desired and type any non-blank character. These tab stops are meaningful during data entry and field modification (:M#) since data within a field may be modified without disturbing the rest of the line. A maximum of 10 tab stops may be set.

:RH - RPG Header - sets tab stops for RPG header specification at columns 6 and 15.

:RF - RPG File - sets tab stops for RPG file description specification at columns 6, 15, 24, 33, 40, 54, 66 and 70.

:RE - RPG Extension - sets tab stops for RPG extension specification at columns 6, 11, 19, 27, 33, 36, 40, 46, 52 and 58.

:RL - RPG Line - sets tab stops for RPG line counter specification at columns 6, 15 and 20.

:RI - RPG Input - sets tab stops for RPG input specification at columns 6, 15, 21, 44, 53, 59 and 65.

:RC - RPG Calculation - sets tab stops for RPG calculation specification at columns 6, 18, 28, 33, 43, 49, 54 and 60.

:RO - RPG Output - sets tab stops for RPG output specification at columns 6, 15, 23, 32, 38, 40 and 45.

:RS - RPG Summary - sets tab stops for RPG summary specification at columns 6, 14 and 23.

:X - TEXT - this command enables word wrap around and disables shift key inversion and space insertion after leading periods. It automatically enters the tab set command (:T), so that tab stops may be cleared by the operator. The tab key character is not changed; therefore, the :<tab key> command must be used to set a new tab key character if one is desired.

:<tab key> - change tab key character to any non-alpha, non-numeric, non-COLON, non-ENTER character typed after a leading colon on the command line.

19.7 Recovery Procedures

A 'FORMAT TRAP' occurs when a record not belonging to the current file is encountered. This can be caused either by a physical misalignment of the disk read head or because a record has erroneously been written into that file by some other program.

A 'RANGE TRAP' occurs when the physical limit of the file is reached and no end of file is present.

19.7.1 Bypassing Errors or End of File

When a format or range error occurs, an appropriate message appears on the command line and the cursor is turned off. In order to proceed, the operator must first press the DISPLAY key. The effect of either a format or range trap is the same as an end of file and no further data will be read from the file.

To read past a format error or past an end of file, use the BYPASS command, :B, repeatedly if necessary.

19.7.2 File Recovery

If the source file is lost (e.g., erroneously KILLed), the scratch file may contain a useful copy. Since the scratch file (SCRATCH/TXT) usually contains a copy of the last file edited, it may be used to recover only that file.

19.8 Glossary

Assembler mode - assumed mode of execution. Tab stops at 9, 15 and 30 (may be changed during execution). The space bar is assumed as the tab key character (this may be changed in parameter list or during execution). Shift key inversion and no word wrap around are assumed. Leading period (.) generates period space (.) for comment lines. Pseudo-ENTER does line-insert.

Command - characters typed at the left edge of the command line following a COLON (:) which have special meaning to the editor.

Command line - the twelfth line of the screen where most data is entered, lines are fetched and commands are typed.

Command separator - the character in a MODIFY command which

indicates what is to be done (> means insert, < means replace and \ or ¦ mean append).

Comment field - in assembler code the area of the screen from columns 30 to 79 which is generally used for programmer comments.

Comment mode - assumed if 'C' in parameter list. Facilitates changing or adding comments to assembler code. Tab stops at 9, 15 and 30 (may not be changed during execution). The space bar is assumed to be the tab key character (this may be changed in parameter list or during execution). Shift key inversion and no word wrap around are assumed. Leading period (.) generates period space (.) for comment lines. Pseudo-ENTER positions to comment field of following line and deletes the comment. Delete and Scratch commands affect only the comment field. Trailing blanks are truncated when data is output.

CONFIGURATION FILE - A file, default name of EDIT/OV1, which automatically provides default options to EDIT.

DATABUS mode - assumed if 'D' in parameter list. Tab stops at 9 and 15 (may be changed during execution). The space bar is assumed to be the tab key character (this may be changed in the parameter list or during execution). Shift key inversion and no word wrap around are assumed. Leading period (.) generates period space (.) for comment lines. Pseudo-ENTER does line-insert. Input lines are blank filled and trailing blanks are truncated on output.

Field number - a digit used in the MODIFY command to designate characters between two tab stops. Field '1' is always from column 1 to the first tabstop; thus, in Assembler mode, '1' designates the label field, '2' the opcode field, '3' the expression field and '4' the comment field. During field modification, trailing fields are preserved.

Format trap - bad record encountered on disk. See 'RECOVERY PROCEDURES'.

Line insert - results from an INSERT command, data entry or modification when word wrap around is in effect or a Pseudo-ENTER key in any mode other than Comment. The lines above the pointed line are rolled up and a new,

blank line is generated at the pointed line.

Logical scratch file - current output file.

Logical source file - current input file.

New text - a group of characters, typed immediately after a command separator in a modify command, which will become part of the line being modified.

Old text - a group of characters, including spaces, which are searched for, either in the pointed line (as in the MODIFY command) or in the file (as in the FIND or LOCATE commands).

One-pass option - assumed if 'O' in parameter list. The one-pass option does not update the physical source file. The FIND, LOCATE and END, END/DEL commands will not write back into the input file if this option is set.

Parameter list - initialization information provided when the editor is first executed. Following file specifications, a SEMI-COLON (;) indicates the presence of a parameter list. The mode, one-pass option, tab character, margin bell column and (in text mode) 'no shift inversion' (S) and 'no word wrap around' (L) may be set.

Pointed line - a pointer (>) in the left hand margin is used to reference lines for modification by command. The line to the right of the pointer is the pointed line.

Physical scratch file - specified (or implied SCRATCH/TXT) output file.

Physical source file - specified input file.

Pseudo-ENTER - the key marked DEL (always shifted) is referred to as the Pseudo-ENTER key. If pressed in the first column of the command line, one line of text will be fetched from the source file.

In comment mode, if pressed on any but the bottom screen line or command line, it will cause the cursor to be positioned to the comment field of the following line and that field will be erased.

In all other modes, the Pseudo-ENTER key causes a new line to be inserted so that data entry may proceed in the

same area of the screen. If pressed on the last screen line, the Pseudo-ENTER key simply places the cursor on the command line.

Range trap - attempt to read past the end of allocated space on the input file - see 'RECOVERY PROCEDURES' in the previous section.

Scratch file - at any point in time, the logical scratch file is the output file. It may, however, physically be the original input or the assigned 'scratch' file.

Screen line - any of the eleven lines on the screen which may be referenced by the command pointer. The command line is not, therefore, included.

Shift key inversion - reverse the function of the shift key for all alpha characters so that, in lower case, alpha characters will appear upper case.

Source file - originally this is the input file specified at initial execution. The term source file refers to the current input file; thus, at any point in time, the logical source file may be either the specified input file or the file specified as the scratch file.

Text mode - assumed by a 'T' in the parameter list. No tab stops are set (tabs may be set during execution). The SEMI-COLON (;) is the assumed tab character (the tab key character may be changed in the parameter list or during execution). No shift key inversion is performed (this may be selected in the parameter list). Word wrap around is performed (this feature may be turned off by an 'L' in the parameter list).

Word - a word is defined as any group of less than 50 characters preceeded by a space.

Word wrap around - a feature of text mode. During data entry a space within the last 10 columns of the screen cause an immediate carriage return. If this occurs on a screen line, a line insert is performed so that data entry may proceed at the same area of the screen. If a character is typed over the last column of the screen, the last word is removed, a line insert performed and the removed word is placed at the beginning of the inserted line where data entry may proceed. If a modify command causes the line to become longer than 79 characters, the

trailing characters, including the last word on the line, will be moved to a new line which will be inserted below the original line. Control will then return to the command line.

19.9 Command List

:A	APPEND pointed line to command line and roll up
:B	BYPASS end of file
:C	COPY pointed line to command line and roll up
:D	DELETE entire line
:D <old text>	DELETE from left thru <old text>
:E	END edit - copy remainder of file and update source
:EO	EOF display - fetch and display data until end of file
:E/	END/DELETE update without copying remainder
:E*	EOF search - find end of file and display last full screen
:F <old text>	FIND match - search file for matching leading text
:F<SPACE>	FIND repeat - use previous find/locate <old text>
:F*	FIND display - display previous find/locate <old text>
:I	INSERT a blank line below pointed line
:L	LOCATE next - clear screen and get next line
:L <old text>	LOCATE match - search file for matching imbedded text
:L<SPACE>	LOCATE repeat - user previous find/locate <old text>
:L*	LOCATE display - display previous find/locate <old

text>

LINE MODIFICATION

:M [old text]<[new text] - MODIFY replace old text by new text, adjusting the entire line

:M [old text]>[new text] - MODIFY insert new text after old text, adjusting the entire line

:M [old text]\[new text] or :M [old text]¦[new text] - MODIFY append new text after old text adjusting the entire line

FIELD MODIFICATION

:M<#> [old text]<[new text] - field MODIFY replaces old text within specified field with new text without disturbing the remainder of the line.

:M<#> [old text]>[new text] - field MODIFY inserts old text after new text within specified field, without disturbing the remainder of the line.

:M<#> [old text]\[new text] or :M<#> [old text]¦[new text] - field MODIFY appends the new text after the old text within the specified field without disturbing the remainder of the line.

:M* MODIFY displays the previous modify [old]<sep>[new]

:M[#] MODIFY repeats the previous modify [old]<sep>[new]

:RH RPG HEADER - sets tab stops at columns 6 and 15.

:RF RPG FILE - sets the stops at columns 15, 24, 33, 40, 54, 66, and 70.

:RE RPG EXTENSION - sets tab stops at columns 6, 11, 19, 27, 33, 36, 40, 46, 52, and 58.

:RL RPG LINE - sets tab stops at columns 6, 15, and 20.

:RI RPG INPUT - sets tab stops at columns 6, 15, 21, 44, 53, 59, and 65.

:RC	RPG CALCULATIONS - sets atab stops at columns 6, 18, 28, 33, 43, 49, 54, and 60.
:RO	RPG OUTPUT - sets tab stops at columns 6, 15, 23, 32, 38, 40, and 45.
:RS	RPG SUMMARY - sets tab stops at columns 6, 14, and 23.
:SB	SCRATCH BELOW deletes the pointed line and all screen lines below it
:SC	SCRATCH ABOVE deletes the pointed line and all screen lines above it
:T	TAB SET permits the user to set up to ten tab stops
:X	TEXT mode switches to text mode with word wrap around and no shift key inversion.
:<character>	changes the tab key character to <character>.

CHAPTER 20. FILES COMMAND

FILES is a program which selectively prints or displays DOS file descriptions in file name sequence.

One may select information pertaining to all DOS files or only those files with names and/or extensions beginning with the characters specified by the operator. Selected directory entries are sorted into ascending file name sequence. If desired, information from associated Retrieval Information Blocks (described elsewhere in this User's Guide) is also extracted for each Directory entry. Extracted data is interpreted and displayed on the screen or listed on a Local or Servo printer.

20.1 Command Description

Program execution is initiated by the operator typing in the name FILES followed by selection criteria and display options (if option codes are to be used):

```
FILES [file-name][/file-ext][:DRn],[<subdir-name>]
  [,<output-file>][;options]
```

file-name:	Select entries for files with names beginning with the 1-8 characters specified.
file-ext:	Select entries for files with name extensions starting with the 1-3 characters specified. This criterion must be preceded by a slash.
DRn:	Specifies the disk drive to be selected. This criterion must be preceded by a colon. If this criterion is omitted, drive 0 will be selected.
subdir-name:	Specifies the named subdirectory from which to select entries.
output-file:	Specifies the disk file to which the selected entries will be written.

options: The following option codes must be preceded by a semi-colon. but may be entered in any order:

N - Suppress file allocation map.
D - Display on CRT.
L - List on local printer.
S - List on servo printer.
F - Write output to disk as DOS text-type file.

If options are keyed and D, L, S and F are omitted, then D is assumed. If F is keyed and the <output file spec> is not present in the command line, one is requested by the message:

DOS OUTPUT FILE SPEC:

20.2 Default Messages

If no option codes are entered, the following messages will be displayed on the CRT:

SUPPRESS FILE ALLOCATION MAP?

If "Y" or "YES" is entered in response to this message, the display of file allocation information from Retrieval Information Blocks (RIB) will be suppressed. If any other response is entered, file allocation information will be displayed for each selected file.

After the user has replied to the map selection message, the program will test to see if there is a servo printer connected to the processor that is ready for printing. If a servo printer is attached and ready, the following message will be displayed:

LIST ON SERVO PRINTER?

If the user enters a "Y" or "YES" in response to this message, the servo printer will be selected to display output. If any other response is entered or the program cannot find an available servo printer, the program will test to see if a local printer is connected and ready for printing. If the program finds that a local printer is available, the following message will be displayed:

LIST ON LOCAL PRINTER?

If the user enters "Y" or "YES" in response to this message, the local printer will be selected for output.

If the program cannot find an available printer, or the operator fails to select a printer with an option code or in response to a message, the program will display file descriptions on the screen.

20.3 File Descriptions

If a printer has been selected for output, the following message will be displayed:

ENTER HEADING:

Up to 32 characters can be entered that will be displayed at the top of each page of listed output.

File descriptions are sorted into ascending file name sequence for easy reference and displayed or printed in the following format:

FILENAME/EXT (PFN) DW

DW flags following the Physical File Number (PFN) indicate if the file is delete protected (D), or write protected (W). If the file allocation map was not suppressed, messages describing the file's size and location will be included in the file description.

Depressing the DISPLAY key during display or printing of file descriptions will cause the program to pause until the key is released. Depressing the KEYBOARD key will cause the program to terminate and return control to the operating system.

20.4 Error Messages

*** PARITY ERROR ***

FILES can not continue due to an irrecoverable parity error encountered while trying to read data from the disk.

*** DRIVE OFFLINE ***

FILES is unable to connect to the disk drive selected by operator (drive 0 if not otherwise specified).

FILE(S) NOT FOUND.

No Directory entries have been found that meet the users selection criteria.

INVALID DRIVE

An invalid drive specification was entered.

CONFLICTING OPTIONS SPECIFIED

Options specify output on more than one device.

UNRECOGNIZABLE OPTION CODE

An unrecognizable code has been entered in the option field.

PRINTER NOT AVAILABLE

An option code specifies a printer that does not respond when tested for status.

CHAPTER 21. FIX COMMAND

FIX <file spec>

This will cause a set of six zeros, two spaces, and three more zeros to be displayed on the bottom line. (The zeros represent the current address and its contents.)

000000 000

The screen is then rolled up. The program is waiting for a command from the operator.

Commands are in the form [number][character] where the number is assumed to be octal. If the number is omitted, a value of zero is used.

The following is a list of command characters with their effect:

ENTER KEY - If no block of object code is currently in memory (as at the beginning or after a block has been rewritten), search the object file forward until a block containing the given location is found, then display the contents of that location.

- If a block of code is in memory and the location given is within the limits of the block, the contents of the location will be displayed.

- If a block is in memory and the location given is not within the block limits, the current address will be set to the minimum or maximum address of that block, its contents will be displayed and a beep will sound.

M - Change the contents of the displayed address to the number given.

I - Increment the current address (up to the maximum address in the current block).

. - Change contents of displayed address to number given and automatically increment the current address and display the contents of that location.

D - Decrement the current address (down to the minimum address in the current block).

T - Transfer the modified block back to disk - rewriting it in place. After the block is written, the current address is set back to zero, so that all searches always start from the beginning of the file.

A - Abort processing the current block, set the current address back to zero.

0 OR * - Return to the operating system - if there is a block of object code in memory, it is <u>not</u> written back into the file.

If the command character is not one of the above, it is ignored and regarded as if only the ENTER KEY had been pressed. If the <filespec> is not an ABS file, the message

RECORD FORMAT ERROR

is displayed.

If the file specified on the command line is not found, the message

NO SUCH NAME

is displayed.

CHAPTER 22. FREE COMMAND

22.1 Purpose

As a disk becomes full, it is frequently useful to know how many 256-byte sectors remain available for allocation. Another useful bit of knowledge on the larger disks is how many empty slots in the directory remain for the allocation of file names. This is precisely the function of the FREE command.

22.2 Use

The FREE command accepts a drive specification. It may be entered simply as:

```
FREE
```

which will cause the FREE space and files for all the on-line drives to be displayed. It may also be entered as:

```
FREE :DRn
```

which will display the FREE space and files for only drive n.

The command scans all drives that it finds on-line and displays (1) the number of available file names (representing possible files to be created) and (2) the number of available 256-byte sectors that it finds on each.

Holding down the "Display" key will cause FREE to pause. Pressing the "Keyboard" key will cause FREE to terminate and return to the operating system.

CHAPTER 23. INDEX COMMAND

23.1 Introductionn

The DOS INDEX command is used to create the tree structure required by programs using the indexed sequential access method (ISAM).

The INDEX command has the capability of creating index files from any DOS text-type files. The indexed access method can then rapidly access records in this file either in sequential or random order. Records in files to be indexed must contain a single record key up to 100 characters long contained in the first 249 bytes of each record.

The format of the key is mmm-nnn where mmm is the beginning character position of the key field in each logical record and nnn is the ending position of the key field. It is required that the second key specification (nnn) be greater than the first specification (mmm). Note that each record must have a unique key.

It is possible to build many independent indices to permit access to records of the same file by many separate, unrelated keys. There are no restrictions on the number of indices that may be built, or on the relationship or lack of relationship among the various keys used.

23.2 System Requirements

INDEX runs under the DOS operating system. In addition, INDEX uses the DOS SORT command, which must be resident on an online disk at the time INDEX is used. If the INDEX command is to pre-process the text file, the REFORMAT command must be available. (See the Section on PREPROCESSING the file).

23.3 Operation

When the Index command is to be executed, the operator must enter:

INDEX <file-spec>[,<file-spec>];<parameters>

where only the first file specification and key field description are mandatory, and specify the file to be indexed. Default extension is /TXT. The second file specification is the name of the INDEX file to be created. If no file is specified, the name of the first file is used with default extension /ISI. If no drive is specified, the INDEX file will be placed on the same drive as the file to be indexed. Note that INDEX files may have any names at all - and be located on physically different drives from the file being indexed.

23.3.1 Parameters

In addition to the parameters that INDEX itself recognizes, the user may specify any parameters acceptable to the REFORMAT utility (if preprocessing is to be done) or a primary record specification to be passed to SORT. Parameters recognized by INDEX are as follows:

F -- Preprocess the input file with REFORMAT
p -- Display the SORT and REFORMAT parameters
(Note that this is a lower case "p")
E -- Index in EBCDIC collating sequence.

The primary record specification is an option that allows the user to create the ISAM index file from a subset of the data file. The format of the primary record specification is Pnnn^C. The P must always appear. The field following P, denoted by nnn, represents the place in each logical record where a one position field exists that differentiates records in the file. The location of this one character field must be less than or equal to 249. The caret (^) can have one of two values. It can be either an equal sign (=) or a pound sign (#). If the former, it means create the ISAM index file from all records that contain the ASCII character C in position nnn. If it is a pound sign, it means that the ISAM file will be created from all records that do not contain the value of C in position nnn.

In general the parameters for INDEX can be specified in any order and may optionally be separated from each other by one or more blanks. The only exception to this is when a primary record specification exists, it must precede the key field specification and be separated from the key by a blank or a comma.

23.4 Choosing A Record Key

Since the speed of access to an indexed file varies according to how much file space and thus how many levels of index are required for the index tree, the choice of what to use for a record key becomes highly important. Of course, you must choose a key which will uniquely determine the record you wish to access, but you should scrupulously avoid including information in the key which is not absolutely necessary. For example, a file could be keyed according to automobile license plate numbers. Typically, these numbers will include a hyphen or other punctuation, which could easily be excluded from the record's key. The indexed access method will perform more efficiently if all non-significant characters are removed from the record's key.

23.5 Preprocessing the File

In file structures such as an indexed file where records are randomly inserted and deleted, the file tends to become non-optimum for searching. In addition, due to the method with which the indexed access method inserts records, each inserted record exists in a separate disk sector. This means that for records that are 80 characters long, two-thirds of the disk space for each additional record is wasted. This results in a reduction of the performance of the indexed access method.

In order to reclaim space vacated by deleted records and padding bytes in inserted records, the file may be processed by the REFORMAT utility prior to indexing.

23.5.1 Invoking Reformat

The INDEX utility will automatically invoke REFORMAT if the "F" option is present when INDEX is invoked. You must have specified the options that REFORMAT will need to process the file.

Note that if multiple indices are to be created, reformatting need only be specified for the first INDEX step, and MUST not be specified later if it was not specified in the first step. Although REFORMAT will not destroy the file, specifying reformatting will invalidate any previously built indices.

Basically, you must tell REFORMAT what format the records of the file are to have after reprocessing. You may select record compression, space and record compression, or blocking. Since the reformatting is done in-place, the REFORMAT option cannot enlarge the file which is to be indexed. For additional details on the

REFORMAT utility, see the REFORMAT section of this guide.

23.5.2 Considerations for Unattended Indexing

Users who use the INDEX command from a CHAIN file (see the section on the CHAIN command for more details) and used AUTOKEY to restart their chain in the event of a failure should generally avoid using REFORMAT directly from INDEX. The reason why is that REFORMAT as invoked by INDEX uses the REFORMAT-in-place mode of the REFORMAT command. (The reason for this is that it is faster to do so, and also allows the indexing with reformatting of a file which is too big to REFORMAT in the available scratch space on a single-drive, almost full disk). Although REFORMAT is very careful not to damage the file being processed, if the file is actually in the process of being reformatted when a power failure occurs, the results can be undesirable.

This potential problem during unattended INDEX chaining can be avoided by setting a checkpoint (see the AUTOKEY command description for details), copying the original file to a scratch file, setting another checkpoint, reformatting the scratch file back into the original (using the COPY mode of REFORMAT), setting a further checkpoint, and finally INDEXing the file using INDEX. In this way there is always an undamaged file with which execution can resume if necessary.

23.6 INDEX Messages

The INDEX command produces several messages on the operator's console. The content and meaning of these messages follow:

FILE PREPROCESSING WILL BE DONE BY REFORMAT COMMAND
: This message indicates that the user has requested preprocessing of his file by the REFORMAT command.

COMMAND STRING ERROR - TERMINATOR MISSING
: This is an internal error - report to DATAPOINT.

REFORMAT/CMD IS MISSING
: The user has requested preprocessing, but the REFORMAT command is not present on disk. You must load it.

INDEX PARM ---->
: This is the parameter string that will be passed to the INDEX overlay and used to build the index file.

REFORMAT PARM ---->

This is the parameter list passed to the REFORMAT command.

INFILE NAME MISSING
This indicates that you have omitted the first, and mandatory file specification. Put it there.

KEY SPECIFICATION MISSING
You have not given index information on the location of the key in the record.

TOO MANY DIGITS IN KEY SPECIFICATION
The key field specification must be no more than 6 digits long.

ERROR IN FIRST COLUMN OF KEY
The first key field specification is invalid.

KEY SPECIFICATION NOT TERMINATED BY 015
The key field specification must be the last field in the parameter string.

SORT MUST BE PRESENT
INDEX has discovered that the SORT command is not resident. It must be loaded.

KEY TOO LONG
The key is over 100 characters in length.

INFILE DISAPPEARED AFTER SORT
This is an internal error - notify DATAPOINT.

TAG FILE NOT GENERATED BY SORT
This is an internal error - notify DATAPOINT.

ILLEGAL CHARACTER IN KEY: XXX
The character whose octal form is displayed was found in a record key. Only ASCII text characters are permitted.

DIGIT PRECEDING PRIMARY FIELD SPECIFICATION
INDEX has found a digit where it doesn't belong - remove it.

PRIMARY SPECIFICATION INVALID
The Primary record specification passed to SORT has invalid syntax.

INVALID TAG RECORD - SOFTWARE OR DISK ERROR
The tag file has an invalid record. Possible hardware fault, notify DATAPOINT.

MORE THAN ONE RECORD HAS THE KEY: key
Duplicate keys exist in the file to be indexed. The offending key field is displayed.

INDEX WILL USE EBCDIC SORT
The user has requested an index using the EBCDIC collating sequence.

LAST COLUMN OF KEY LESS THAN FIRST COLUMN OF KEY
The first key field specification must be less than the second specification

23.7 ISI File Formats

The DOS indexed file structure consists of a multi-level radix tree structure based on the record keys, and contains pointers to the location of the keyed records. Note that since many of these pointers are physical disk addresses, the ISI file cannot be moved without re-invoking INDEX. The text file may be moved so long as it is unchanged in any way. Moving the ISI file will destroy it.

The different levels of indices all have the same content, except for the lowest level index. Index levels are built up until the highest level of index will fit in a single disk sector. This requirement is the reason for the 100 character limitation on key length.

The ISI files have the following format:

Offset	Length	Description
000	003	PFN and LRN bytes as per DOS convention - see the chapter on SYSTEM STRUCTURE.
003	0nn	This is a KEY entry where nn is key length+7 for a lowest level index, and key length+3 for a higher level index. The first sector of an ISI file after the RIBs is a special header record. Note that as many key entries are put in a sector as will fit without splitting across a sector boundary.

Each KEY entry for a higher level index has the following format:

Offset	Length	Description
000	KEYLEN	The highest key in the next lower level index sector.
KL	001	Octal 012 - This indicates the end of the key and that this is a higher level index entry.
KL+1	002	Sector and Cylinder of the entry in the next lower level of index.
KL+3	001	Octal 0377 - This indicates that this is the last entry in this sector.

Each KEY entry for a lowest level index entry has the following format:

Offset	Length	Description
000	KEYLEN	The key for this particular record.
KL	001	Octal 015 - This indicates that this is a lowest level index entry and delimits the end of the key.
KL+1	003	Buffer Offset, address for the logically next lowest level index entry.
KL+4	003	Buffer Offset, and logical record number of the text file record having this key.
KL+7	001	Octal 0377 - Indicates that this is the end of the lowest level index.

The first data sector in an ISI file is a header record used to locate the file from which the index was built. In this way, it is only necessary to specify the name of the index to DATASHARE.

Offset	Length	Description
000	003	PFN and LRN indicators as per DOS convention. See DOS Advanced Programmer's Guide (Part IV).
003	013	Name of the data file that goes with this index file.
016	003	PFN, RIB sector, and RIB cylinder of this file. This field is used to check that the index file has not been moved.

021	003	PFN, RIB sector, and RIB cylinder of the file indexed.
027	003	Buffer address and LRN of the last record used in the data file.
032	003	Buffer address and LRN of the first free index entry.

23.8 Examples of the use of INDEX

First, a simple example in which only a single ISI file is created, with the same name and on the same device as the text file it indexes. The file is a list of bad checks presented at a local grocery chain, and now each store has a DATASHARE terminal to inquire on the current status of each deadbeat. Thus, while the file is accessed often, additions and deletions are fairly infrequent, so the file will not be reformatted. The file is keyed by bank number (8 digits) and account number (7 digits) concatenated and in positions 1 to 15 of each record.

In order to create (or recreate) the index file, the operator must type:

```
INDEX DEADBEAT;1-15
```

The INDEX program will then create a file DEADBEAT/ISI which DATASHARE can use to access the DEADBEAT/TXT file.

Now, this same grocery chain has expanded its operations, so it desires to include more information on the location and date of each NSF check presented. Therefore, they have expanded the file to include the old key in positions 1 to 15, a store location number in positions 16 to 18, and a date field in positions 19 to 24. As an afterthought, the manager decides to tack on the name of the person passing the bad check in positions 193 to 216.

In order to create the indices required for access by any of these keys, the operator must type:

```
INDEX DEADBEAT,BANK;1-15
INDEX DEADBEAT,DATE;19-24
INDEX DEADBEAT,STORE;16-18
INDEX DEADBEAT,NAME;193-216
```

The INDEX program will create four files with names BANK/ISI, DATE/ISI, STORE/ISI, and NAME/ISI. Each file is logically separate, yet all are on the same volume as DEADBEAT/TXT.

Now the store owners have uncovered a hitch - first, the number of bad checks is becoming so large, there is no room on one disk for all the index files and the text file. In addition, access has been slowing way down as the frequency of additions and deletions increases. The store owners have called DATAPOINT to complain, and their local systems engineer has told them they need to reformat the files when they re-index, and has sold them another disk drive.

The operator now types:

```
INDEX DEADBEAT,BANK/ISI:DR1;F1-15
INDEX DEADBEAT,DATE/ISI:DR1;19-24
INDEX DEADBEAT,STORE/ISI:DR1;16-18
INDEX DEADBEAT,NAME/ISI:DR1;193-216
```

Note that the reformatting is done only once at the beginning. While it does no harm to reformat each time, it will waste much time and accomplish nothing. If reformatting had not been done when the first index was built, it could not be correctly done later without invalidating the previously built indices.

CHAPTER 24. KILL COMMAND

KILL - Delete a file from the directory

KILL [file spec]

The KILL command deletes the specified file from the system if the file is not protected. If the file is protected in any way, the message

NO!

will be displayed. If the file specification is not given on the command line (file names which contain special characters cannot be given on the command line), the request for the file name:

```
WHAT FILE?  EXAMPLE:  SCRATCH /TXT:DR1
          /    :DR
```

will appear. The user must keyin an eight character filename (including trailing spaces), a slash, a three character extension (including trailing spaces), a colon, the letter "D" and the drive number on which the file resides. If the entire filename specification is not entered properly, the message:

NO SUCH NAME.

will appear. If the specified file cannot be found (both a name and an extension must always be supplied if specified on the command line), the message:

NO SUCH NAME.

will be displayed. If the file is found and is not protected, the message: THAT FILE IS <filename> ON DRIVE n will appear. Then the operator must additionally answer the message:

ARE YOU SURE?

with a 'Y' before the actual deletion of the file is achieved. After the deletion has occurred the following message is displayed:

* FILE DELETED *

CHAPTER 25. LIST COMMAND

25.1 Purpose

The LIST command will list any DOS standard format text file on the screen, a local or servo printer.

The command can be used for such things as:

A quick scan of a file by displaying it on the screen (LISTing a file is faster than EDITing it);

Producing a hardcopy listing of a file for permanent records;

Listing a file for use in preparation of a BLOKEDIT COMMAND FILE.

In this Section, the following terms apply:

Text file means a file with records containing only ASCII characters, except for space-compression bytes and the End-Of-Record and End-Of-File marks. Files created by EDIT and those produced by DATASHARE are normally in the class of text files.

Line means one record of a text file. When displayed on the screen, only the first 72 characters of a record will be displayed; when listed on a local or servo printer only the first 124 characters will be printed. (The remaining eight characters contain a line number.)

Record means the user logical record number (LRN). The first LRN of a file is zero.

25.2 Parameters

When the LIST program is to be executed, the operator must type:

LIST <filespec> [,spec2][,filespec2][;[O][,X][,F][,I][,Nn]]

The square brackets ([]) indicate optional fields and the pointed brackets (<>) indicate a required field.

25.3 INPUT File Specification

The file specification (<filespec>) must refer to a DOS text file. If no extension is supplied with the file specification, an extension is assumed depending on the options given. The default extension of TXT is assumed unless the option "I" or "F" is used. The option "I" (list a file using its index) has a default extension of ISI and the option "F" (list a file with format control bytes) has a default extension of PRT. If no drive is supplied with the file specification, all drives will be searched for the filename/ext. If <filespec> is omitted, the message

NAME REQUIRED.

is displayed. If the file indicated by <filespec> is not found on an online volume, the message

NO SUCH NAME.

is displayed.

25.4 Starting Point

The operator may specify a line number, or logical record number, in the file at which the list should begin by including an optional second parameter [,spec2] with the file specification. For example:

LIST <filespec>,L400

would list the specified file beginning with the line 400 of the file. If the line number specification exceeds the number of lines in the file, LIST returns to DOS after displaying the message:

FILE EXHAUSTED BEFORE LINE FOUND.

LIST <filespec>,R18

would directly access logical record 18 of the specified file and list, starting at line number 1. If range or format errors occur, the error type is indicated and another record number is requested.

For instance, if the record number specification exceeds the number of records, the message

RANGE - NEXT RECORD NUMBER:

is displayed.

The DEFAULT value for the second parameter is line 1 and record 0.

25.5 OUTPUT File Specification

If the options "P" (write to a print file on disk) or "Q" ('QUEUED' write to a disk print file starting at the end-of-file mark) are used, then the third parameter (filespec2) may be used to specify the output file. If the filename is not given, it is assumed to be the same as the input file name. If the extension is not given, it is assumed to be PRT.

25.6 Output Device

The operator may specify an output device [O] other than the CRT display by including an optional parameter of "S" (servo printer), "L" (local printer), "P", or "Q". For example:

```
LIST <filespec>,L400;S
```

would list the specified file on the Datapoint servo printer starting at line 400 or

```
LIST <filespec>;L
```

would list the specified file on a Datapoint local printer beginning at line number one.

The DEFAULT output device is the CRT display which may be specified by entering a "D".

25.7 Output Format

A parameter [X] is available to suppress line numbers. If the 'X' is entered, lines of up to 132 characters will be printed. For example:

```
LIST <filespec>;SX
```

would put the output on the servo printer without line numbers, whereas,

```
LIST <filespec>
```

would put the line numbered listing on the screen.

25.8 Format Control

The parameter [F] is available to allow the handling of print files (those with a format character in the first column of each line). If 'F' is entered, the file will be listed without line numbers, page numbers, or headings. The following characters cause the following action to be taken before the line is printed.

1 - Skip to top of form

\+ - Suppress line feed

(space) - Single line feed

0 - Double line feed

\- - Triple line feed

Any other character in the first column will be handled as a space (single line feed) and discarded.

25.9 Operator Controls

The listing consists of a continuous stream of the listed file's text, preceded by the line's number in the file. To cause the listing to pause, the operator may hold down the DISPLAY key. To abort the listing, the operator may depress the KEYBOARD key.

If the output device is the local or servo printer, the output will be listed at 54 lines per page (unless "Nn" is entered as an option where n is the number of lines to a page) on continuous form paper, with each page numbered and titled by the file specification and an optional heading. The heading is entered by the operator when the LIST program displays the message:

ENTER THE HEADING:

before printing begins. The number of each line in the file will be printed at the left margin of the page.

CHAPTER 26. MANUAL COMMAND

MANUAL - Clear Auto Execution

MANUAL

If the auto-execution name has not been set the message

AUTO NOT SET.

will be displayed. Otherwise, the System Table location reserved for the auto-execution information will be cleared and the message

AUTO CLEARED.

will be displayed.

CHAPTER 27. MASSACRE COMMAND

27.1 Purpose

The MASSACRE command is provided to ease the user's job of removing all files from a scratch disk. It deletes all files on the specified logical drive without regard to whether or not delete or write protection is set. When MASSACRE completes, the only files remaining on the MASSACREd drive are the eight system files, SYSTEM0/SYS through SYSTEM7/SYS, as well as CAT/CMD and MIN/CMD.

27.2 Use

MASSACRE <drive spec>

Before the specified disk is MASSACREd, the user is asked several times to acknowledge his request before actual deletion of the files begins. A typical console dialog would look something like the following:

```
MASSACRE :DR2
KILL ALL NON-SYSTEM FILES ON :DR2? Y
I'M KILLING nnn FILES, nn OF THEM ARE PROTECTED.
ARE YOU SURE? Y
REALLY? Y
```

After the MASSACRE operation completes, only the ten files specified above will remain on the MASSACREd drive. Note: Users should consider regenerating the disk in lieu of using MASSACRE. MASSACRE maintains locked out areas of the disk but regeneration provides a thorough check of the disk during its process. The opportunity to recheck the disk should not be overlooked.

CHAPTER 28. MIN COMMAND

28.1 Purpose

The Multiple In (MIN) command is useful for reading multiple files (source, object, and Datashare object) from the front cassette drive to disk. It will handle all standard single file (OUT and SOUT), double file (SOBO), and multiple file (LGO, CTOS, and MOUT with or without a directory) tape formats.

28.2 Tape Formats

Multiple In will automatically process the tape format by the following conventions if an option is given.

28.2.1 Single File Tapes

An OUT (object out) tape format has a file mark zero, a file mark one, an object file with entry point, and a file mark 0177. An object file has an address with the MSB and LSB in the fourth and fifth bytes of each record. Their complements are in the sixth and seventh bytes. The remainder of each record is filled with octal characters (ranging from 0 to 0377).

A SOUT (source out) tape format has a file mark zero, a source file, a file mark one, and a file mark 0177. A source file consists of records containing only ASCII characters, except for space compression bytes, physical end-of-record bytes, and logical end-of-record bytes.

28.2.2 Double File Tapes

A SOBO (source and object out) tape is the combination of a SOUT and OUT tape. It has a file mark zero, a source file, a file mark one, an object file with entry point, and a file mark 0177.

28.2.3 Multiple Numbered-File Tapes

An LGO (load and go) tape has a loader, a file mark zero, a string of files (the first being an object file and the rest may be source, object, Databus Code, and Relocatable Code intermixed) separated by sequential file marks, and a file mark 040.

A MOUT (multiple out) tape without directory has a file mark zero, a string of files (may be source, object, and Datashare object intermixed) separated by sequential file marks, and file marks 040 and 0177. Single and double file tapes are included in this category if options are not used.

28.2.4 Multiple Named-File Tapes

A CTOS (cassette tape operating system) tape has a loader, a file mark zero, a CTOS object file with entry point, a file mark one, a catalog object file, a string of files separated by sequential (though not necessarily contiguous) file marks, and a file mark 040.

A MOUT (multiple out) tape with directory has a file mark zero, a tape directory, a string of files separated by sequential file marks, and file marks 040 and 0177. The directory is a source format file containing a date entry seven bytes long (DDMMMYY) and 31 file name entries each eleven bytes long (eight bytes for the name and three bytes for the extension). The entries are separated by end-of-string bytes (octal 015). This makes it convenient for display under CTOS LIST or to load to disk and list.

28.3 Parameters

28.3.1 Single File Tapes

For OUT, and SOUT tape formats, the file specifications may be included on the command line in the following manner:

MIN [<file spec>];<option>

where <option> is an 'S' for SOUT tape formats.

File specifications are of the form FILENAME/EXT:DR#. If the drive is not given, all drives online will be searched starting at drive zero. If the extension is not given, the assumed extension

(TXT, ABS, DBC, or REL) will depend on the file format. MIN will identify the tape format. If the file name has not been entered on the command line, the program will ask:

LOAD FILE #XX (format)?

where, XX indicates the file number on the cassette and format indicates the type of file (SOURCE OBJECT, DATABUS CODE, or RELOCATABLE CODE). If the file is to be loaded, the response Y (yes) will cause the message:

DOS FILE NAME:

to be displayed on the same line. If the response is N (no), the operator will be asked for the next file (if any). If the response is *, control is returned to DOS. If no name is entered, the message:

NAME REQUIRED

will appear. If the filename specified already exists, the message:

NAME IN USE. WRITE OVER?

will appear. The answer N (no) will cause the filename request to be displayed again. The answer Y (yes) will cause the disk resident file to be overwritten. If the file to be overwrittten is write protected, the message:

WRITE PROTECTED OVERWRITE?

will appear. If the response is not Y, the filename request will be displayed again. If the response is Y, the protection is changed from write protect to delete protect and the disk resident file is overwritten. When a file has been loaded from the cassette the message:

LOADED

will appear to the right of the filename. The message:

MULTIPLE IN COMPLETED

indicates the successful completion of the program.

28.3.2 Double File Tapes

The file specifications for a SOBO tape may be entered on the command line in the following manner:

MIN [<file spec>][,<file spec>];B

File specifications are of the form discussed above. If the second file name is not given, the first name with the assumed extension of ABS will be used. If the extension is not given with the first name, TXT will be assumed. If the filename has not been entered on the command line, MIN will operate in the same manner as described in the section on single file tapes above for each file on the cassette, displaying the messages in the same order for both files.

28.3.3 Multiple Numbered-File Tapes

LGO tapes and MOUT tapes without a directory are both handled in the same manner. MIN is first executed as:

MIN

An LGO tape will then be identified as:

LGO TAPE FORMAT

In the case of multiple files, MIN will operate in the same manner as described in the section on single file tapes above for loading a file without entering the name on the command line. The questions described will be asked for each file on the tape until end of file has been encountered on the tape or an * is entered in response to the "load" question. MIN bypasses the loader on a LGO tape before searching for the file. If the file is not found, the message:

FILE NOT FOUND

will appear and MIN will be terminated. If the file is found and the file name is not entered on the command line, the file name will be requested as in single-file tapes.

28.3.4 CTOS Tapes

A CTOS tape will be identified as:

CTOS SYSTEM TAPE FORMAT

The system then searches for the catalog (tape file #1). The CTOS file is fairly long so it takes a while. If the catalog file is not an object file or is an object file that loads into memory somewhere other than 017406 or 017410, the message:

BAD CATALOG

will appear and the remainder of the tape will be processed as a multiple numbered-file tape starting at tape file #2. If a good catalog is found, it will then be displayed as:

CATALOG: <file 1> <file 2> <file 3> <file 4>. . .

Then the operator will be asked:

DO YOU WANT TO LOAD <file 1> ?

The entire process is identical to the multiple numbered-file tapes above except the first fourteen files are referred to by name. The filename may be expanded by the operator from the six character name allowed by CTOS to the eight character name allowed by DOS plus the extension. A filename is requested if the reply is 'Y'.

28.3.5 MOUT With Directory Tapes

These tapes are processed in a manner very similar to CTOS tapes. The tape is first identified as:

MOUT TAPE FORMAT

Next the date will be displayed:

DATE: DD MMM YY

Then the directory will be displayed:

DIRECTORY: <file 1/ext> <file 2/ext> <file 3/ext>. . .

Then the operator will be asked:

LOAD <file 1/ext> ?

All the responses are the same as above except that the file name will not be requested. A drive response is also available. Entering "DRn" will imply "YES" and force the file to drive n. The program will cycle until the end-of-tape file mark (040 or 0177) is read at which point the message:

MULTIPLE IN COMPLETED

will be displayed.

28.3.6 Options

Tape file modifications may prevent MIN from automatically determining the tape format. In this event, the options 'L' (for LGO), 'C' (for CTOS), or 'D' (for Directory) are available. Also, option 'N' (for No directory) will tell the system that it is handling a MOUT tape without a directory which allows entering the file names manually if the directory entry names are not desired. This also allows entering the directory to disk. A drive specification is also available. Entering a "Dn" or "DRn" will force drive n to be assumed. Options are entered following a semi-colon.

These options are merely test overrides. If a tape, for instance, starts with a recognizable file mark, a loader won't even be tested for and therefore entering the 'L' option is meaningless.

Unfortunately, MIN cannot differentiate an OUT, SOUT, or SOBO tape from a MOUT without directory tape. To speed the processing, the options 'S' (for SOUT) and 'B' (for SOBO) are available. Once again, if the tape doesn't resemble a SOUT tape, for instance, entering an 'S' is meaningless.

If the tape is a MOUT tape with a directory, the options 'A' (for All), 'O' (for Overwrite), 'Q' (for modifying the extension with Q's) and :DRn (for loading the files to Drive n) are available. Using the option 'A' will load all the files. However, if the file already exists, the operator will be asked if overwriting is desired and if not, for a new file name. Entering the 'O' option in conjunction with the 'A' will force overwriting of existing files (unless write protected). If while processing in the 'All Overwrite' mode a write protected file is encountered, the message:

WRITE PROTECTED

will appear and processing continues with the next file. Entering the 'Q' option in conjunction with the 'A' will put as many Q's into the directory extension as necessary to create a new filename/ext if the original one already exists. If the original filename/ext exists, the message:

EXISTING FILE

will appear to the right before the modification to the extension is performed. If the filename/QQQ already exists, the message:

Q OPTION EXHAUSTED

will appear to the right and the file will be skipped.

The option 'N' followed by an octal number allows that specific file to be loaded. For example, entering:

MIN FILE/TXT;N12

will load the tape file following file mark 12 (octal) to disk as 'FILE/TXT'. The default extension will be 'TXT' for source, 'ABS' for object, 'DBC' for Databus Code object files, and 'REL' for Relocatable Code files depending on the tape file format. If a non-octal number is entered (e.g. N8) the message:

NUMBER NOT OCTAL

will appear and MIN will be terminated. If an unrecognizable record format is encountered, the message:

UNRECOGNIZABLE TAPE RECORD FORMAT

will appear and MIN will be terminated. MIN bypasses the loader on a LGO tape before searching for the file. If the file specified is not found, the message:

FILE NOT FOUND

will appear and MIN will be terminated. If the file is found and the file name is not entered on the command line, the file name will be requested as in single-file tapes.

The options 'L', 'C', 'N', 'S', 'T', and 'B' are mutually exclusive. Only one may be entered. The 'A' may be entered with or without the 'D' and with none of the other above options. 'O' and 'Q' are mutually exclusive and may only be entered in

conjunction with the 'A'. If any of these restrictions is violated or a character other than those above entered, the message:

BAD OPTION PARAMETER

will appear and the program will be aborted.

28.4 Errors

If the tape format is not one of the eight standard formats outlined above in Section 23.1 (e.g. it starts with a file mark two) the message:

INVALID TAPE FORMAT

will appear and the processing will be aborted. If the end of tape is detected while processing, the message:

END OF TAPE

will appear and the processing will be aborted. If a parity error is encountered in an object or Datashare file on tape, the message:

PARITY ERROR-FILE DELETED

will appear, the file name will be removed from the disk directory, and processing will skip to the next file. If a parity error is encountered in a source file on tape, the message:

PARITY ERROR-RECORD MODIFIED

will appear, a 253 byte disk record will be written with percent signs in the first five positions of the record data, and processing will be continued with the next record.

CHAPTER 29. MOUT COMMAND

29.1 Purpose

The Multiple Out (MOUT) command is useful for writing multiple (up to 32, or 31 if a directory is used) disk files (source, object, and Datashare) out to the front cassette drive.

An additional feature is the ablity to create a tape file directory as file #0 on the tape. The directory is a source format file, that is, it consists entirely of ASCII characters except for space compression bytes, physical end-of-record marks, and logical end-of-record marks. The directory contains a date entry seven bytes long (DDMMMYY) and 31 file name entries each eleven bytes long (eight bytes for the name and three bytes for the extension). The entries are separated by end-of-string bytes (octal 015). This makes it convenient to list under CTOS LIST or to load to disk and list. The directory is also used by the MIN program to enter files to disk. MOUT creates the directory in memory before the tape writing starts even if it is not to be written to tape. The writing of a full tape (over 500 records) takes about 10 minutes which shows the advantage of entering all the names before writing begins.

Another feature is the option to automatically verify a tape following its creation. Or a previously written directory tape may be verified in an 'only verify' mode. If this is requested, the system will read the directory on the cassette tape in the front drive (if a valid directory is not found, the system will abort with the appropriate message) and verification will be performed against the indicated files.

29.2 Parameters

File specifications and/or options may be entered on the command line in the following manner:

MOUT [<file spec>,<file spec>,...][;options]

File specifications are of the form FILENAME/EXT:DR#. If the drive is not given, all online drives will be searched starting at drive zero. If the extension is not given, ABS is assumed. File specs are separated by anything (including multiple spaces) except

letters, numbers, slash (/), or colon (:).

29.3 Options

Options (which follow a semi-colon and may be spaced or separated by commas) are 'L' for a loader format tape, 'D' for a directory format tape, 'V' for verification of the created tape, and 'X' for verification only.

If a loader is to be written, the first file (file #0) must be an object file. There are no restrictions on files other than #0.

The directory option ('D') will write a tape directory as file #0. The first item within the directory is the date entered DDMMMYY. Note: the month is entered as three alpha characters. The date may be entered following the option letter (e.g. D 12JAN74). If the date is not entered, it will be requested.

The verify option ('V') will verify all the files on the created tape. Verification consists of making a byte for byte comparison between the data on the disk and the data on the tape. If verification fails, the tape will be rewritten and verification tried one more time.

The verify only option ('X') will cause the first tape file to be read from the front deck. If it is a directory (first seven characters of DDMMMYY format), the remaining files will be automatically verified using the directory entries. If it is a loader, it will be verified and file names requested for the remaining files as they are verified. An 'N' may be entered immediately preceding the 'X' to force the system not to recognize the directory. This would be done if manually entering file names is desired (for instance, the directory names don't match the disk file names). If there is neither a directory or loader, file names are requested as the files are verified.

If the semi-colon is entered with no entry following, it will be interpreted that the tape will not have a loader, a directory, or any verification.

Entering 'D' and 'L' together or entering something with 'X' or entering some letter other than 'D', 'L', 'V', or 'X' will result in the message:

BAD OPTION PARAMETER. MOUT DISCONTINUED.

and the Multiple Out will be aborted.

If file names and/or options are not entered on the command line, MOUT will ask for them as required. If options were not entered, the first question will be:

DO YOU WANT A LOADER?

Replies other than 'Y' or 'N' will be answered by:

WHAT?

and a repeat of the question. If the reply is 'N', the next question is:

DO YOU WANT A DIRECTORY?

Again, if the reply is other than 'Y' or 'N', it will be answered by:

WHAT?

and a repeat of the question. If the reply is 'Y', the next request is:

ENTER THE DATE (DDMMMYY):

where the month is entered as three alpha characters. If the day is not in the range of 00 to 39, the month not alpha, or the year not in the range of 70 to 99, the response:

BAD DATE

will appear and again the request for the date. The next question is:

DO YOU WANT TO VERIFY THE TAPE?

If the reply is not 'Y' or 'N', the response:

WHAT?

will appear followed by a repeat of the question. If the reply is 'Y' and the replies to the loader and directory questions are 'N', the question:

DO YOU WANT TO ONLY VERIFY THE TAPE?

will then be asked. If the reply is other than 'Y' or 'N', the response

WHAT?

will appear followed by a repeat of the question. If only verification is requested, the first tape record on the front tape deck is read in. If it is a directory (the first seven characters of DDMMMYY format), the remaining tape files will be automatically verified using the directory entries. If it is a loader, the message:

LGO TAPE FORMAT

will appear. The message:

LOADER IS BEING VERIFIED

will then appear as the loader is being verified. If the loader verifies correctly, the message:

LOADER OK

will appear to the right. Otherwise, the message:

BAD LOADER

will appear. After checking the loader or if the tape has neither a loader or directory, the message:

CASSETTE FILE #XX (format) DOS FILE NAME:

will appear where XX is the file number and (format) is (SOURCE), (OBJECT), (DATABUS CODE), or (RELOCATABLE CODE) depending on the file format. If nothing is entered, the message:

NAME REQUIRED

will appear and the request repeated. If an asterisk (*) is entered, MIN will terminate and return to DOS. If a greater-than sign (>) is entered, the program will skip to the next file. If a less-than sign (<) is entered, the program will backspace to the prior file (bypassing null files). If the program finds the beginning of the tape, it will beep and then move forward to the first file. If a name is entered, the default extension is 'TXT' for source, 'ABS' for object, and 'DBC' for Datashare object depending on the file format. If the drive number is not entered, all online drives will be searched starting at drive zero. If a drive number greater than DOS allows is given, the message:

BAD DRIVE

will appear and the request repeated. If the file is not found, the message:

FILE NOT FOUND

will appear and the request repeated. If the disk file is found, it will be matched byte by byte against the disk file. If the files completely match, the message:

FILE OK

will appear to the right and processing continues with the next file. If an error is detected, the appropriate message will appear and processing continues with the next file. Null files are bypassed. Processing continues until an end-of-tape mark (file mark 040 or 0177) is read at which time the message:

VERIFICATION PHASE COMPLETED

will appear and MOUT will be terminated.

29.4 File Names

If the file names are not given in the command line, the operator will be asked for the file names one at a time. The request is of the form:

CASSETTE FILE XX DOS NAME:

where XX is the file number. Possible replies to the file name query include:

a) the file specifications as discussed above,
b) a pound sign (#) which will bump the file number to 20 octal if not already there (only allowed on loader tapes to initiate numbered files on a CTOS tape),
c) a dollar sign ($) which will cause a null file (tape file mark only) to be written to tape and the file spec of NULL/NUL to be entered in the directory,
d) an asterisk (*) which will indicate no more files are to be entered and the tape writing started (writing is postponed until the directory is complete), and
e) OS which will abort the program. The message: MULTIPLE OUT DISCONTINUED will appear and control is returned to DOS. (To dump OS/ABS, enter 'OS/ABS').

If the operator fails to enter a name, the message:

NAME REQUIRED

will appear and the name request will be repeated. If the drive is given and is not in the range valid for DOS, the message:

BAD DRIVE

will appear followed by a re-request of the name. If the file is not found, the message:

FILE NOT FOUND

will appear followed by a re-request of the name. If the file is found, the format (object, source, or Datashare) will be determined by the system. If the tape is a loader tape and file #0 is not an object file, the message:

FILE FOLLOWING LOADER NOT OBJECT

will appear along with a re-request of the file name. This message may also be displayed if the reply to the file name query for file #0 is a pound sign. Otherwise the messages:

OBJECT FILE

or:

SOURCE FILE

or:

DATABUS CODE FILE

or:

RELOCATABLE CODE FILE

or:

NULL FILE

will appear to the right of the file name. If the pound sign is entered for a tape that does not have a loader, the message:

NOT LGO TAPE

will appear with a re-request of the file name. If 32 files (or 31 on a directory tape) are entered, the message:

THAT'S THE END OF THE LINE

will appear and the tape writing is started automatically.

29.5 Writing

Once the tape writing has started, the system will keep the operator informed of its progress. As a loader is being written, the message:

LOADER IS BEING WRITTEN

will appear. As a directory is being written, the message:

DIRECTORY IS BEING WRITTEN

will appear. While files (including null files) are being written, the message:

FILE <filename/ext> IS BEING WRITTEN

will appear. When the writing is completed, the message:

WRITING PHASE COMPLETED

will appear.

If a non-object record is sensed in an object file while writing to tape, the message:

FILE CONTAINS NON-OBJECT RECORD

will appear and the next file is written over the bad tape file including the file mark. This will leave a directory entry without a file. If this should happen, it will cause verification to display the message:

NON-SEQUENTIAL FILE MARK

and the tape rewritten.

If a non-source record is sensed in a source file while writing to tape, the message:

INCORRECTLY FORMATTED SOURCE RECORD

will appear. The file is ended at this point without writing the bad record and the next tape file will start immediately following. If this should happen, it will cause verification to display the message:

INCORRECTLY FORMATTED DISK RECORD

or:

TAPE EOF BEFORE DISK EOF

and the tape rewritten.

If MOUT runs out of tape, the message:

END OF TAPE ENCOUNTERED WHILE WRITING filename/ext

will appear, an end of tape marker written at the end of the previous tape file, and the unwritten files will be removed from the directory (if there is one). Processing then will be continued with verification.

29.6 Verifying

If verification is requested, the system will keep the operator informed of its progress. As a loader is being verified, the message:

LOADER IS BEING VERIFIED

will appear. As a directory is being verified, the message:

DIRECTORY IS BEING VERIFIED

will appear. While files (including null files) are being verified, the message:

FILE filename/ext IS BEING VERIFIED

will appear. When the verification is completed, the message:

VERIFICATION PHASE COMPLETED

will appear. If verification is requested for a tape having no directory, the message:

NOT DIRECTORY TAPE

is displayed. Then the message:

CASSETTE FILE #XX(format) DOS FILE NAME:

will appear. The filename should be entered. Responses are discussed in the section under OPTIONS.

A variety of error messages may be displayed during the verification phase. Most of them are self-explanatory. They include:

BAD LOADER

BAD DIRECTORY

TAPE FILE DOES NOT MATCH DISK FILE

INCORRECTLY FORMATTED DISK RECORD

DISK FILE CONTAINS NON-OBJECT RECORD.

DISK FILE CONTAINS NON-TEXT RECORD.

NON-SEQUENTIAL FILE MARK.

TAPE FILE MARK READ BEFORE TAPE OBJECT EOF.

TAPE OBJECT EOF NOT FOLLOWED BY TAPE FILE MARK.

DISK EOF BEFORE TAPE EOF

TAPE EOF BEFORE DISK EOF

If an error is detected, the program will then either rewrite the tape (if it has just been created) or skip to the next file (if in the 'verify only' mode). If it rewrites the tape, the message:

I'M NOW REWRITING THE TAPE

will appear. The system will rewrite once before quitting completely at which point the message:

VERIFICATION UNSUCCESSFUL

will appear and the processing terminated.

If a problem arises that causes an abnormal end (e.g. end of tape), the message:

MULTIPLE OUT DISCONTINUED

will appear, otherwise the message:

MULTIPLE OUT COMPLETED

will signal the successful end of the program.

ERROR D ON DECK 2

will signal parity errors on the cassette and control is returned to DOS.

CHAPTER 30. NAME COMMAND

NAME - Change the name of a file

NAME <file spec1>,[<file spec2>][,<subdirectory name>]

NAME will allow the user to change the name of a file, the extension of a file, or the subdirectory in which a file resides. The content of the file is unchanged. The first file specification refers to the current file name and the second file specification is the new name and/or extension to be assigned. If no extension is supplied in the first file specification, ABS is assumed. If no extension is supplied in the second file specification, the extension of the first file is assumed. If no extensions are supplied, both files will be assumed to have extensions of ABS. The drive number should only be specified in the first file specification.

If the NAME command is used to move a file from one subdirectory to another the second file specification may be omitted (unless the filename and/or extension are to be changed) and the subdirectory name denoting the subdirectory into which the file is to be placed is the third specification:

NAME <file spec1>,,<subdirectory name>

In both uses of the NAME command, two specifications are required. If either name is not given, the message

NAME REQUIRED.

will be displayed. If the second name is already defined on the drive that contains the first file, the message

NAME IN USE.

will be displayed. Note that the drive specification on the second name is ignored. If the first name is not found on an online disk, the message

NO SUCH NAME.

will be displayed. If the subdirectory name keyed is not found

on the disk containing the file to be renamed, the message

NO SUCH SUBDIRECTORY.

will be displayed. If the third parameter is not specified, the file is "brought into" the current subdirectory at the completion of the renaming process.

CHAPTER 31. REFORMAT COMMAND

31.1 Introduction

The DOS REFORMAT command is used to change the internal disk format of text-type (non-object) files. Additionally, it can recover disk space left unused when files are updated by the DATASHARE indexed sequential access method. REFORMAT can compress a file in place on disk provided that such compression does not entail the writing of a physical disk sector prior to the time that sector is read. REFORMAT maintains logical consistency in such cases and will not write on a disk file until it has checked to be sure it can complete its job successfully.

31.2 Operation

When the REFORMAT program is to be executed, the operator must type:

REFORMAT <file-spec>[,<file-spec>][;<parameters>]

where only the first file specification is mandatory, and specifies the file to be reformatted. If the second file specification is given, it must be distinct from the first. Reformatting in place is requested by omitting the second file specification.

The parameter list describes the format the output file is to take, and whether REFORMAT is to free any disk space that might be vacated by the reformatting process. In addition, the user can specify that REFORMAT is to pad short records, and either truncate or segment long records. Reformat will produce three different kinds of output files: record compressed, space and record compressed, and blocked records (See the section on TEXT FILE FORMATS.). Note that REFORMAT will not produce blocked space compressed records or space compressed non record compressed files although such files can be used as input to the REFORMAT program.

The valid parameters that can be passed to REFORMAT are as follows:

Parameter	Description
B<n>	The output file will be blocked. This implies no space or record compression, with <n> logical records per physical sector.
C	The output file will be space and record compressed. The number of logical records per physical sector will be indeterminate.
R	The output file will be record compressed, but no space compression will be done. In general, the number of logical records per physical sector will be indeterminate.
L<n>	The length of each logical record will be adjusted to <n> characters. Note that if the logical records are space compressed, this will not make the physical length of the records <n> characters. If the logical record is shorter than <n> characters, it will be padded with blanks to the proper length. If the logical record is longer than <n> characters, the action taken depends on the T and S parameter.
T	(Only valid if L parameter is given) Truncate the logical record if it is longer than <n> characters.
S	(Only valid if L parameter is given) If the length of the logical record is greater than <n> characters, segment it into (q) logical records each of length <n>, padding if necessary. The number (q) is defined as input length divided by <n> rounded upward to the next integer. If neither S or T is specified, and an input record of length greater than <n> is found, a message is issued and REFORMAT gives up.
D	If reformatting is done in place and this parameter is specified, any disk space vacated by the reformatting process will be returned to the operating system for re-use.

31.3 Output File Formats

The REFORMAT utility permits you to select essentially three different output file formats. It will produce blocked files that are not space compressed, record compressed files that are not space compressed, and files that are both record and space compressed. In addition, it has a subcommand to permit you to specify the logical length of the output records. Use of this subcommand will guarantee that each record has exactly the same logical length. Note that if the output format does not specify space compression, the physical length of each record will be identical. This is especially useful for telecommunications disciplines that require records of fixed length.

If you have set a fixed logical length for output records, there are two subcommands available to tell REFORMAT what to do with records whose logical length exceeds the specified output length. You may select either truncation of the input record, or you may segment it into two (or more) output records, each of the logical length specified.

31.4 Reasons for Reformatting

Several uses of REFORMAT deserve special mention. First, a random disk file is structured to have one logical record per physical sector. Often, however, it is convenient to create a random file through the use of the general purpose editor - which record and space compresses its output. REFORMAT can then reprocess the file into the correct format for DATASHARE or DATABUS random access.

Secondly, when a file is accessed with DATASHARE indexed sequential access method, any additions or deletions result in an increase in the physical size of the file. The reason for this is that any inserted records are placed at the physical end of the file, and each one consumes at least one entire physical sector, regardless of its logical length. Similarly, deleted records are simply overstored with octal 032 (logical delete) characters, and the space they vacate is not reused. REFORMAT recognizes this condition, and will recover such vacated space. Note that ISAM read-only or update-only (no additions or deletions) files do not usually need reformatting.

31.5 Reformat Messages

The REFORMAT utility program produces several messages on the operator's console. The contents and where necessary, meaning of those messages follow:

REFORMAT VERSION 1
Self-explanatory.

COMMAND LINE ERROR
This is an internal error and should be reported to Datapoint.

PROGRAM ERROR - EXCESS FILE SPACE NOT DEALLOCATED TO PREVENT POSSIBLE LOSS OF DATA
REFORMAT has detected an invalid end of file mark. In order to prevent the possible loss of data which might be after the invalid end of file indicator, space allocated but unused is not freed.

EXCESS FILE SPACE NOT DEALLOCATED; OUTPUT FILE IS DELETE PROTECTED.
Self-explanatory.

OUTPUT FILE IS WRITE PROTECTED AND CANNOT BE WRITTEN INTO OR SHORTENED.
You have requested REFORMAT to output to a write-protected file.

INVALID OPTIONS SPECIFIED
You have given REFORMAT an invalid parameter list. This message is followed by the valid options you may specify.

ILLEGAL CONFLICTING OPTIONS
You have specified two mutually exclusive options.

YOU SPECIFIED BOTH SEGMENTATION AND TRUNCATION, YOU CANNOT HAVE BOTH
Self-explanatory.

BLOCKING FACTOR CONTAINS ILLEGAL NON-NUMERIC DIGITS
Self-explanatory.

BLOCKING FACTOR REQUIRED BUT MISSING OR ZERO
   You specified blocking but omitted the blocking factor.

LOGICAL RECORD LENGTH REQUIRED BUT MISSING OR ZERO
   You must specify the logical record length of the output file if you wish to have fixed length output records.

YOU HAVE ILLEGALLY ENTERED A SPECIFICATION FOR A THIRD FILE
   REFORMAT recognizes only two file specifications.

HOW DO YOU EXPECT TO FIT THAT MANY RECORDS IN A 256 BYTE SECTOR?
   Self-explanatory.

LOGICAL RECORD LENGTH, IF SPECIFIED MUST BE <= 250 BYTES.
   Self-explanatory.

YOUR BLOCKING FACTOR IS TOO LARGE FOR THE SIZE OF THE RECORDS YOU HAVE.
   Self-explanatory.

YOUR LOGICAL RECORD LENGTH IS TOO SMALL FOR THE SIZE OF THE RECORDS YOU HAVE
   While processing the input file, REFORMAT came across a record that was larger than the specified logical record length. Since you specified neither segmentation nor truncation, this is recognized as an error.

SPECIFIED OUTPUT FILE FORMAT ENLARGES PRESENT INPUT FILE. INPUT FILE CANNOT BE ENLARGED DURING REFORMAT-IN-PLACE. REFORMAT IN-PLACE REQUEST REFUSED.
   Self-explanatory.

YOU SPECIFIED AN OUTPUT FILE THAT ENDED UP BEING YOUR INPUT FILE. TO REFORMAT IN-PLACE DON'T SPECIFY ANY OUTPUT FILE.
   Self-explanatory.

INPUT FILE IS EMPTY!
   You are attempting to reformat a null file.

OUTPUT FILE NOT FOUND ON DRIVE X.
OUTPUT FILE FOUND ON DRIVE Y.
OUTPUT FILE WILL BE CREATED ON DRIVE Z.
These messages only occur if no specific drive was indicated for the output file. The first message appears followed by either the second or third. REFORMAT could not find the output file on the same drive as the input file. It either found one on a different drive, or created one on the displayed drive. If the output file is created, it is always created on the same drive as the one the input file is on.

REFORMAT IN-PLACE REQUESTED.
PRESCAN IN PROGRESS.
REFORMAT is checking to make sure it can properly process the file inplace.

FILE WAS ALREADY IN THE SPECIFIED FORMAT
Self-explanatory.

COPYING WITH REFORMATTING IN PROGRESS
Self-explanatory.

INPUT FILE NAME REQUIRED
Either you gave only an extension or drive for the input file, or you specified the output file first, followed by the input file.

INVALID DRIVE SPECIFICATION
The drive number was greater than allowed or you did not specify the drive in the form :DR<n>.

31.6 Text File Formats

Under Datapoint Corporation's Disk Operating System, text files consist of legal ASCII characters, which make up the text itself, and various special control characters with special meanings. It is illegal to have the control characters in the text portion of the file. According to DOS convention, any character between 000 and 037 is considered a control character.

Each physical record of a text file is a logical disk sector, and contains 256 characters. The first three and last two characters are reserved for control functions; hence, the maximum space available in a single physical record is 251 bytes. The format of a logical sector is as follows:

Offset	Length	Description
000	001	Physical file number of this file. For a detailed description of physical file organization, see the DOS Advanced Programmer's Guide.
001	002	Logical record number. This refers to logical physical records, and is not related to text records within the file.
003	373	Text. 251 bytes of text and control characters, depending upon the format of the file.
376	002	Two characters reserved.

The text part of each file is considered a logical stream, crossing sector boundaries without being logically discontinuous. Demarcations of logical record boundaries are made solely by control characters imbedded within the text itself. There are essentially five control characters found in files generated by DOS: 000 <NUL> used for end of file indication, 003 <EM> used to denote the end of medium (a sector boundary) but not the end of a logical record, 011 <CMP> used to denote space compression, 015 <ENT> used to denote the end of a logical record, and 032 <DEL> used to denote deleted data.

Under DOS each file is treated as a single, continuous stream of data. Physical records bear no relation to the logical structure of the data contained in them. In this way, a proliferation of different file structures, and the special routines needed to treat such special cases has been avoided. This does not mean that there cannot be a relation between physical and logical structure, it simply means that such a relationship is incidental to a particular file, and need not be treated as a special case. For example, random access to a data file is defined in the DATABUS language. Files to be accessed in this manner are structured in such a way that one logical record corresponds exactly with one physical record. This structure is

not inherent in the makeup of a random file, in fact, such files can be treated exactly as any other text file.

The basis for this treatment of text files is the logical record. A logical record starts at the beginning of a file, or immediately after the end of a previous logical record. It consists of ASCII data and is of no pre-determined length. Instead, the record is terminated with a single ENT character. In this way, complications arising from a multitude of record types are entirely avoided.

If the logical record contains any CMP characters, it is said to be space-compressed. The character immediately following the CMP character is a space count, and the pair represent the number of ASCII blanks removed when the record was compressed. Since the character following CMP is always assumed to be a space count, CMP can never occur as the next-to-last text character in a physical sector, since the EM character following it would be lost.

If the file is organized so that each physical sector contains exactly the same integral number of logical records, with no logical record spanning an EM character, the file is said to be blocked. If the file is not blocked, then it is said to be record compressed. Note that for a blocked file all sectors except possibly the last one in the file contain the same number of logical records while for record compressed files the number of logical records per physical sector is indeterminate.

Under DOS conventions, a valid end of file mark consists of exactly six NUL characters, followed by an EM character:

000 000 000 000 000 000 003

This mark must begin at a sector boundary. All information after a valid end of file mark in the sector is indeterminate.

CHAPTER 32. REWIND COMMAND

REWIND - Rewind the cassette tape.

REWIND [REAR or DECK1]

The cassette in the front deck is rewound unless "REAR" or "DECK1" is specified. If no cassette is in place in the deck, the rewind will proceed but only after a cassette is put into place. The cassette can be fully wound onto the clear leader at the very end of the tape, since the rewind command starts by slewing the tape backwards for a few seconds first. This both takes up any slack that may be present in the cassette before the high-speed rewind starts, and also ensures that the tape is not on the clear leader when the actual rewind begins.

CHAPTER 33. SAPP COMMAND

SAPP - Append two source files creating a third

SAPP <file spec>,[<file spec>],<file spec>

The SAPP command appends the second source file after the first and puts the result into the third file. If extensions are not supplied, TXT is assumed. The first two files must exist. If the third file does not already exist, a new file will be created. The first file's end of file record is discarded and the copy is terminated by the end of file mark in the second file.

Omitting the second file specification causes the first file to be copied into the third file. Note that neither the first or second file is changed.

The first and third file specifications are required. If either is omitted the message

NAME REQUIRED

will be displayed.

The second and third file specifications must not be the same.

CHAPTER 34. SORT COMMAND

34.1 Introduction

The Disk Operating System SORT enables any Datapoint Disk user to initiate file sorts directly from the keyboard.

Using a multi-train radix sort technique, the Datapoint processor achieves speeds comparable with much larger systems. The list of options also compares favorably with much more extensive systems. Nevertheless, since it uses the full dynamic nature of the Disk Operating System, it is extremely easy to operate. (Users who have spent several hours figuring out how to set up the myriad of SORT work datasets required, even for the simplest sorts, by other sort packages know what we're talking about.)

For more sophisticated uses, SORT may be called from other programs through CHAIN. Using CHAIN also enables complicated sort options to be reduced to a single file name then callable either from the keyboard or another program. CHAIN also extends the SORT package to operate as a merge, as well.

34.2 General Information

SORT will optimize its speed through allocation of its working files on the available drives. During this process it attempts to ascertain the availability of sufficient disk space to achieve the desired sort. The program will abort at this point should the disk space be inadequate.

34.3 Fundamental SORT Concepts

34.3.1 File formats

All Datapoint systems use a universal text file structure recognized by Databus, Datashare, RPG II, Basic, Scribe, Editor, Assembler, Terminal emulators, etc. Therefore, any text file generated by or for any of the above, may be sorted. The file to be sorted must be on disk, however.

There are two sub-formats a Datapoint file can have: Indexed or Sequential. Notice that throughout the SORT section of the User's Guide, "Indexed" refers to direct random, as opposed to ISAM, access (see INDEX). Indexed files are required to have a single 'string' or 'record' of data per physical disk record. SORT assumes indexed files have space compression. This implies that the logical position of a character in a record and the physical position of a character in a record may differ. The SORT will always expand the spaces to determine the logical position of a character. The maximum record size for indexed records is 250 bytes. Sequential records have no fixed relationship to physical disk records and are written as densely as possible in the given file space. Nonetheless, indexed files can be read sequentially in the identical way that sequential files are read. In fact, both types of files, when read sequentially, are indistinguishable. Indexed files are used for achieving random access to records. They generally require more disk space than sequential files for the same amount of data.

When sorting, consider that the result of the sort is not a restructuring of the original file. It is a NEW file which is a restructured COPY of the original file. The original file is never changed.

Therefore, SORT produces a file which is a sorted version of the original. This gives the user the added opportunity of specifying the type of file to be output regardless of the input file format (with one restriction - see the section on INPUT/OUTPUT FILE FORMAT OPTIONS).

34.3.2 The key options

The KEY of a sort is the FIELD or that part of the record which is to ORDER the sequence of records. For instance, it can be a person's name, state, employee number, amount in debt or any aspect of the data base identifiable by a fixed position in the record based upon the column count from the beginning of the record.

Consider the following record (column count scale below for reference only):

```
Mule, Francis A.        242219 123 BARN    SAN ANTONIO      TX
12345678901234567890123456789012345678901234567890123456789012345678901234567890
```

The name begins in column 1 and goes to 22. The employee number spans columns 24-29. The street address is 31-42. The city is

43-58. The State is 59-60

If each person had a record in the file exactly in the above format, SORT could order the sequence of records in the file by any of the above fields. For instance, to get an alphabetical list of the records by name, the KEY would be 1 to 22 (hereafter referred to as 1-22). The KEY for sequencing the file in order of employee number would be 24-29. The key for ordering the records by state then city and then employee number would be 59-60,43-58,24-29.

It should be obvious that any part of the record can be used as a key. It may not be obvious, however, that the larger the key, the slower the sort - it is, however, the case and it is just about proportional.

34.3.3 How to sort a file

Sorting a file is done from the keyboard of the DOS. All the operator must know is the name of the file to be sorted, the name desired for the sorted output file, and the columns containing the KEY.

For instance, the keyboard issued command for the above example to sort on the name field (1-22), would be:

```
SORT EMPLFILE,SORTFILE;1-22
```

This is assuming that the name of that file was EMPLFILE. It is also the operator's decision as to what the resultant sorted file is called, as the command could have easily been:

```
SORT EMPLFILE,EMPSORT;1-22
```

as well. The second file named is where the resultant sorted output will be placed.

More complicated keys may be stated as well and the command to sort by state and then name would be:

```
SORT EMPLFILE,SORTFILE;59-60,1-22
```

That is all there is to simplified sorting.

Testing SORT for yourself is simple. Most systems have a source code file for a Databus or Assembly language program on the disk. Such programs can be sorted by op-code and provide an interesting analysis of the usage of each instruction type:

SORT INFILE,OUTFILE;9-12

34.4 The Other Options

34.4.1 Generalized Command Statement Format

The following is the generalized statement format for the Datapoint DOS SORT:

```
SORT IN,OUT[,:DRk][,SEQ][;[[F][O][R][H][GNNNTC][N]][K1]...[,On][,Kn]]
```

Information contained within a pair of square brackets [] is optional; information within brackets is order-dependent. Commas may be used to delimit parameters. (NOTE that commas MUST be used to delimit sort-key groups.) The first four fields (those ahead of the semi-colon) are considered to be file specification fields. The fields following the semicolon are considered to be sort key parameters. Default conditions are listed below. Typical statements obeying this format are:

```
(1)  SORT INFILE,OUTFILE
(2)  SORT INFILE,OUTFILE;1-3,7-20
(3)  SORT INFILE,OUTFILE;ID1-3
(4)  SORT INFILE,OUTFILE;IDL7-20
(5)  SORT INFILE,OUTFILE;LH11-20
(6)  SORT INFILE,OUTFILE,,SEQFILE
(7)  SORT INFILE,OUTFILE,:DR0,SEQFILE/SEQ:DR1
```

All the above statements will invoke a sort. Each will provide different results. However, notice that in (1) there are no other parameters than the file specifiers. That is because all the specifiable parameters have a given value in case there is no specification for it.

The following list defines the parameters which can be specified:

IN.....This specifies the input file. This file must exist on disk.

OUT.....This specifies the output file. This specification is optional IF AND ONLY IF the 'L' AND 'H' options are used. If an output file is specified AND no disk drive is

specified AND the file exists on a drive on-line to the system then the output file will over-write the existing file. If an output file is specified AND no disk drive is specified AND no file of that name exists on a drive on-line to the system THEN a file of the given name will be created on the same drive as the input file.

:DRk.....This specifies the drive for the sort key file. This is only a working scratch file needed during the sort. SORT will usually pick the optimum drive on which to put the work file on a multi-drive system. Experience or special considerations may cause the user to want to specify a work drive.

SEQ.....NON-ASCII COllATING SEQUENCE FILE
This specifies the file which contains the collating sequence to be used. If omitted, ASCII will be assumed.

F.....FORMAT.
This parameter specifies the <u>output</u> file format: indexed or space compressed (standard editor output format). If the user specifies I (and the input file is also indexed), then the output file will be left indexed.

Without typing the 'I', the output file will be space and record compressed no matter what the input file. IF AND ONLY IF the input file is an INDEXED file, you may include the 'I' parameter and cause the output file to be indexed.

Note that Indexed in this content refers to physically random accessed (1 logical record per sector) as opposed to ISAM-access files.

O.....ORDER.
This parameter specifies the output file collating sequence: Ascending or Descending. The actual character entered is 'A' or 'D'. The default value is 'A'.

Without typing the 'D', the collating sequence order is considered ASCENDING. Including the D parameter will cause the collating sequence to operate in DESCENDING order. Note that if some keys are to be sorted in ascending order and other keys in descending order, the "On" specification described below should preceed each key whose order differs from the order of the key preceeding it. However, if all keys are to be ordered in the same sequence, this parameter need only be specified once.

R.....RECORD FORMAT.
This parameter specifies a special output record format: Limited output file format or Tag file output. The actual character entered is 'L' or 'T'. The default value is NO SPECIAL OUTPUT RECORD FORMAT; that is, neither 'L' nor 'T', so that the records in the output file will be exact copies (FULL IMAGE RECORDS) of the records in the input file.

Normally the sort transfers all of the records of the input file to the output file. It is possible, not only to transfer part of each record, but to include constant literals in each record as well. Including the 'L' parameter in the list of parameters will cause another question to be asked wherein you may specify the limitations and constants. See the section on Limited Output Format Option.

By entering the 'T' character an output file is generated which consists only of binary record number and buffer byte pointers to the input file records. See the section on Tag File Output Format Option.

H.....HARDCOPY OUTPUT.
This parameter specifies that the output of the SORT will be listed on a printer. The actual character entered is 'H'. The default value is NO HARDCOPY OUTPUT.

Without typing the 'H' no printing will

occur and SORT will require that an output file be named. If the 'H' parameter is given AND an output file is named then SORT will list the output to a printer AND will generate an output file. If the 'H' parameter is given and NO output file is named then SORT will list the output to a printer and no disk file output will be generated.

IF the 'H' parameter is given THEN the 'L' parameter MUST precede the 'H' parameter.

SORT will print to a local printer or a servo printer. See the section on HARDCOPY OUTPUT OPTION.

G.....GROUP INDICATOR

This parameter specifies that the input file consists of PRIMARY and SECONDARY records and specifies which GROUP is to be sorted. The actual character entered is 'P' for PRIMARY or 'S' for SECONDARY. There is no default value.

IF the 'G' option is entered THEN the NNNTC options MUST ALSO be entered.

In a file with PRIMARY and SECONDARY records a string of records with a PRIMARY record as the first record and SECONDARY records following it is considered one block, or group, of records.

When the file is sorted on PRIMARY records the output file has the blocks of records re-ordered so that the PRIMARY records are in the sorted sequence; no change is made in the sequence of the secondary records following each PRIMARY record. When the file is sorted on SECONDARY records and the first key specified is in ascending sequence, the output file has the blocks of records in the same order as in the input file, but the SECONDARY records within each block are in the sorted sequences.

When the file is sorted on SECONDARY records

and the first key specified is in descending sequence, the output file has the blocks of records in reversed order as the input file, but the SECONDARY records within each block are in the sorted sequence.

SORT has no provision for the sorting of PRIMARY AND SECONDARY records in the same SORT run.

NNN.....NUMERIC position of PRIMARY/SECONDARY flag. This parameter specifies the character position for the character (the 'C' parameter) indicating whether the record is a PRIMARY or SECONDARY record . The number MUST be specified if the option is taken and must fall in the range 1 to 249.

T.....TYPE of evaluation.
This parameter specifies equivalence or inequivalence of the group indicator character; that is, whether the character in the record will be EQUAL to or NOT EQUAL TO the character specified. The actual character entered is '=' for equal or '#' for not equal. There is no default character, '=' or '#' must be given if the option is taken.

If '=' is given then if the character in the NNNth position of an input file record is EQUAL to the group indicator character -- indicated by 'C' below -- then the record is a member of the specified sort group -- indicated by 'G' above. Otherwise, it is not a member of the specified group.

C.....CHARACTER, group indicator
This parameter specifies the actual test character for determination of a record's membership in the sort group. The actual character entered is any member of the available character set -- this means any combination of eight bits -- except 015. There is no default character: the character immediately following the 'T' parameter is taken to be the 'C' parameter -- except a 015.

N.....This parameter specifies no space compression on output. This applies to FULL IMAGE and LIMITED OUTPUT files. It does not apply for INDEXED or TAG files.

K1.....SSS-EEE
This is the first sort key specification. If no key is specified, the SORT will assume 1-10,i.e. the first ten characters of the record.
SSS is the starting key position.
EEE is the ending key position. The key is limited to 100 characters and must be contained within the first 249 characters of the record.

On.....This specifies the order for the nth key (ascending and descending are indicated by 'A' or 'D'). If omitted the order used on the previous key is assumed.

Kn.....SSS-EEE
The nth sort key specification. The maximum number of keys is that which can be typed without exceeding the input line.

34.4.2 Keys-overlapping and in backwards order

The key specification need not be only forward. A specification of 17-12 will cause the 6 delimited characters to be a key but in the order of 17,16,15,14,13,12. This is extremely valuable, clearly, in data which has the most significant digit or character last.

Key specifications may also be overlapping: 1-20,30-15 overlaps 15 to 20. When this occurs, the system will optimize the sort and save time over re-sorting on those columns again.

34.4.3 Collating Sequence File

By specifying a sequence file, the user may substitute any collating sequence for the standard ASCII character set. The file name contains eleven characters, eight of which are the file name and three of which are the extension (example, EBCDIC/SEQ:DRn). The last three characters (the extension) must be "SEQ". If the disk drive number on which the file resides is omitted, SORT defaults to the same drive from which the SORT itself was loaded. This table may be supplied by the user but must meet certain

requirements to be loaded:

1. It must be an absolute object file.
2. It must begin loading at location 027400.
3. The first eleven bytes must contain the file name and the extension must be SEQ.
4. The table itself must begin loading at location 027400 and occupy 256 bytes (overstoring the file name described in 3). For instance, the source for the EBCDIC sequence file begins:

```
SET     027400
DC      'EBCDIC  SEQ'
SET     027400
DC      0,1,2,3,4,5,6,7,
.
.
.
```

5. If the file is not found on the specified disk drive the following message is displayed:

   SEQUENCE FILE NOT FOUND

6. If the file is found but is not an absolute object file the following message is displayed:

   SEQUENCE FILE FORMAT ERROR A

7. If the file format appears valid, the file will be loaded using DOS routine LOADX$. LOADX$ will return an error code if the load is unsuccessful. The following display will notify the user of the error:

   SEQUENCE FILE FORMAT ERROR n

   where n=0 if file does not exist
   1 if disk drive is off-line
   2 if directory parity fault
   3 if RIB parity fault
   4 if file parity fault
   5 if off end of physical file
   6 if record of illegal format

34.4.4 Ascending and Descending sequences

Changing the collating sequence from ascending to descending is the same as 'reversing' the file, or placing the last first, etc. Sorting a telephone directory in ascending sequence on name produces the familiar order. Should it be sorted in descending sequence, then Mr. Zyk would be first and Mr. Aardvark would be last. The order of collation, when alphabetic, numeric, and punctuation characters all can occur in a column together, follows the character set order. The sequence may be specified for each sort key. However, it need not be specified if it is the same as the key which preceeds it. Therefore, it is possible to sort portions of the key in ascending order and portions in descending order.

34.4.5 Input/output file format options

SORT accesses each file sequentially. Due to the techniques used in the Datapoint standard file structure, the sequential reading technique will provide SORT with all of the records in the file whether the file was originally indexed or sequential. Therefore, the file format options only allow specification of the output file's format.

If the input file is INDEXED, that is one logical record or string per physical disk record, then you have a choice of output formats. If 'I' is chosen, that is INDEXED, then each output disk record will contain an exact copy of the appropriate input file record. If 'S' is chosen, that is SEQUENTIAL, then the input file, reordered, will be reblocked and appear, generally much more compactly, in the output file in space-compressed sequential format.

If the input file is SEQUENTIAL in its original format, then there is only one choice for the output format; the output file format for a sort on an input file which is sequential MUST be SEQUENTIAL.

34.4.6 Limited output format option

In many cases, especially when making reports, directories etc. from the data base, it isn't necessary to have the entire record transferred from the input file to the output file during a sort. For instance, an entire personnel data base can be sorted by name to produce an internal company telephone directory. However, it is obvious that all that is needed is the name and telephone number, NOT all the other payroll information.

Therefore, SORT permits transferring only that part of the data base desired.

The following is the generalized statement format for the limited output specification which is entered as a second line of parameters:

<(SSS[-EEE]^*^'QQQ')[/(P^NNNTC)]>[,<DUPLICATE OF PRECEEDING>]...

Where different items within parentheses are separated by ^. Only one item within a pair of parentheses may be specified. Items within square brackets [] are optional and items within corner brackets<> may be repeated and must be separated by commas.

The following list defines the parameters which can be specified:

SSS.....STARTING position within input record.
EEE.....ENDING position within input record.
These parameters specify the character positions within the input record to be copied to the output record. The EEE specification is optional; if it is not specified then only one character, the character at SSS, will be copied from the input record to the output record. The SSS and EEE options must fall in the range 1 to 249.

*.....ASCII TAG output.
This parameter specifies that an ASCII pointer to the input record appear in the output record. The ASCII pointer points to the input file logical record number and the byte in that physical disk record containing the first byte of the input file logical record. If the 'I' parameter was specified in the SORT options then, since the byte in the physical disk record containing the first byte of the input file logical record will always be '1', the '1' will not appear. The ASCII pointer is a DATASHARE compatible, leading-zero and space-compressed ASCII number. The number of digits for the logical record number pointer is five; the largest number that can be represented is 65,535. The number of digits for the byte pointer (if

it is generated; that is, the 'I' parameter was not specified) is three; the largest number that can be represented is 250.

QQQ.....QUOTED character string.
This parameter specifies an actual string of quoted characters that is to be copied into the output record. The quoting symbol is the single quote ' mark. The string may include any characters except the ' mark itself and 015, and must be less than 90 characters long.

P.....PRIMARY record to be source.
This parameter specifies that the information specified by the prior set of START/END positions is to be extracted from the primary record for the current record block, rather than the present (secondary) record. This parameter has no effect when an output record is being generated from a primary record.

NNN.....NUMERIC position of evaluation character.
This parameter specifies the character position for the character (the 'C' parameter below) indicating whether the information specified by the prior set of START/END positions is to be copied from the input record to the output record. The number must fall in the range 1 to 249.

T.....TYPE of evaluation.
This parameter specifies the equivalence or inequivalence of the evaluation character; that is, whether the character in the input record should be EQUAL to or NOT EQUAL to the evaluation charater. The actual character entered is '=' for equal or '#' for not equal. If the evaluation is satisfied, then the information specified by the prior set of START/END positions will be copied to the output record.

C.....CHARACTER, record evaluation.
This parameter specifies the actual test character for record evaluation. The actual character entered is any character except 015.

In the same manner that the key of the records is specified by fixed column number, i.e. 1-10 for the first ten characters, the limited output feature specifies that part of the records to be transferred. Should the response 1-10 be given to the limited output format request, only the first ten characters of each record will be transferred to the output file. The limited output format specifier operates in the same manner as the specification of multiple discontiguous sort key fields. For instance, 1-10,50-70 would transfer thirty-one characters from each record of the input file to the output file. The eleventh character in the output record would be the fiftieth character of the input record, etc.

To invoke the limited output format option, the operator includes the 'L' parameter in the specifier list. If and only if the L is specified during the SORT call, will there be a second question asked of the operator on the next line:

LIMITED OUTPUT FILE FORMAT:

This question requires at least one non-trivial field specification or constant(see next paragraph). The number of field and constant specifications is only limited by that which can fit on the keyed in line.

To permit even more utility in report generation, SORT allows inclusion of constants in the output record that didn't occur in the input record. For instance, assume that the personnel data base was a full record of about 240 characters and that the employees name appears in columns 80 to 110 and his telephone number was in columns 171 to 180. To make a telephone directory in alphabetical order, one could answer the following to the limited file output format request:

80-110,' - ',171-180

Note that this would put out the name followed by one space, a hyphen, one more space and the number. Any number of input file fields and constants can be placed in the output file up to the limit of the line on which the specification is typed.

Also note that the output file requires proportionally less room than the input file when limited. Often this fact

can be put to use when the disk file space is nearly exhausted and a sort is required.

34.4.7 TAG file output format option

For some applications it is useful to have a data file sorted into several different sequences. However, to have several copies of a file on disk merely to have it in different sequences consumes a lot of disk space, and indeed if the file is a very large file many copies of it may not fit onto one or even four disk packs.

This problem could be avoided if there were a way to index into the one main file in any of several different sequences. The index pointers could exist as a file, and the index entry for each record in the main file would only have to be three bytes long -- two bytes for the LRN (Logical Record Number) and one byte for the BUFPTR (Buffer Pointer -- a pointer to the beginning of the actual desired record within the disk physical buffer).

SORT provides for the generation of such an indexing file, a TAG file, by the 'T' variation of the 'R' option. A TAG file may be generated for either a Sequential or Index file, and will have the same format for either file. The format of a TAG file is simple:

1. For each record in the input file, the TAG file will have a three byte binary pointer to the first byte of the record.

2. The format of the pointer is:
   Byte 1: MSPLRN (Most Significant Portion of LRN),
   Byte 2: LSPLRN (Least Significant Portion of LRN),
   Byte 3: BUFPTR (Buffer Pointer).

3. The three-byte binary pointers are blocked 83 to a physical disk record.

4. The Physical-End-Of-Record mark is an 003 and the rest 000's.

5. The End-Of-File mark is: beginning at the first byte in the physical record, six 000's, one 003, and the rest 000's.

TAG files may be used by assembly language programs, by RPG II (as Record Address files), and by some Datapoint utility programs, such as the INDEX utility.

For users writing their own Assembly language code to use a TAG file, it is important to know that the MSPLRN and LSPLRN are

together a 16-bit binary pointer to the DOS LOGICAL RECORD NUMBER of the input file, as opposed to the USER LOGICAL RECORD NUMBER. The difference is this: The DOS LOGICAL RECORD NUMBER of a file points to the actual Nth record (starting with zero, the primary RIB) in the file, whereas the USER LOGICAL RECORD NUMBER of a file points to the Nth DATA RECORD (starting with the zeroth data record) in the file. Thus a DOS LRN of zero points to the very first record of the file, which is the master copy of the RIB, a DOS LRN of one points to the second record of the file which is the RIB copy, a DOS LRN of two points to the third record of the file (which is the FIRST DATA RECORD of the file and the USER LOGICAL RECORD NUMBER zero), and so on. The LRN given in the TAG file can NOT be used with the POSIT$ routine unless it is biased by -2. It is much easier to simply place the LRN from the TAG file directly into the LOGICAL FILE TABLE ENTRY for the file that is indexed.

The case with the BUFFER POINTER byte is similar to the LRN pointer bytes. The BUFFER POINTER byte from the tag file is the DOS BUFFER POINTER as opposed to the USER BUFFER POINTER. The difference is this: the DOS BUFFER POINTER points to the actual Nth byte of a disk buffer (starting with zero), whereas the USER BUFFER POINTER points to the Nth DATA BYTE in the disk buffer; the beginning (zeroth) DATA BYTE in the buffer is the fourth byte in the buffer; the first three bytes are reserved for the DOS. Thus, a DOS BUFPTR of zero points to the very first byte in the buffer, which is the PFN (Physical File Number) of the file, a DOS BUFPTR of one points to the second byte in the buffer, which is the DOS LSPLRN, a DOS BUFPTR of two points to the third byte in the buffer, which is the DOS MSPLRN, a DOS BUFPTR of three points to the fourth byte of the buffer (which is the very first DATA BYTE in the buffer), and so on. The BUFPTR given in the TAG file can NOT be used with the GETR$ or PUTR$ routines unless it is biased by -3. It is much easier to simply place the BUFPTR from the TAG file directly into the LOGICAL FILE TABLE ENTRY for the file that is indexed.

If the TAG file option is specified then the LIMITED OUTPUT FILE FORMAT or the HARDCOPY OUTPUT can NOT be specified.

If a TAG file is generated when the 'P' (PRIMARY SORT) option is specified then TAG file pointers will be generated only to the PRIMARY records in the input file.

If a TAG file is generated when the 'S' (SECONDARY SORT) option is specified then TAG file pointers will be generated that point to each PRIMARY record of the input file (in their original sequence) each primary tag being followed by pointers to the

SECONDARY records in the record block in their sorted sequence.

When a TAG file is generated for 'P' or 'S' sorts, no indication is given in the TAG file pointer as to whether the pointer points to a primary or a secondary record; it is up to the user's program to check the records in the indexed file to determine when a record block begins or ends.

34.4.8 HARDCOPY output option

Many times it is desired to have a hardcopy (printed) output from a SORT instead of or in addition to the creation of a disk output file. This can be easily accomplished with SORT by specifying the 'H' (HARDCOPY) option along with the 'L' (LIMITED OUTPUT STRING) option. The 'H' option is essentially an expansion of the 'L' option because disk data files are almost never suitable for full image output to a printer; decimal points need to be inserted into dollar and cents amounts, dashes need to be inserted into part numbers, and spaces need to be placed between dollar amounts and part numbers to colunmnate the data, and so on. If it is desired to list output records in full image format, it is only neccessary to give:

```
1 - n
```

(where n is the maximum printable character on printer) as the limited output string specification.

Sort will not send a line of over 132 characters to a printer. If the limited output specification designates a longer output record, then the full specified formatting will be applied to the disk output file (if any), but only the first 132 characters of the record will be printed.

If the following special characters are imbedded in the output record, they will be interpreted as indicated:

```
015 = End-Of-Record and Carriage-Return/Line Feed.
012 = Line Feed.
014 = Form Feed.
```

SORT will support either a local printer (address 0303) or a servo printer (address 0132). If a servo printer is on-line at the beginning of the FINAL MERGE then it is used as the output printer device; else a local printer will be used. If both printers are available on a system, selection between one or the

other cannot be forced by parameterization; if output is desired to the local printer then the servo printer must be turned off.

34.4.9 Primary/Secondary sorting considerations

If the 'P' (PRIMARY) or 'S' (SECONDARY) SORT option is used then the input file must have a PSPSPS.... format in order for SORT to work as expected, where P is one primary record and S is one or more secondary records. The first record in the file should always be a primary record, and the last record should be a secondary record. There should always be at least one secondary record following each primary record. Tertiary and further level records cannot be accommodated by SORT.

In some cases it may be possible to successfully sort a file using the 'P' or 'S' options even if the file does not faithfully follow the above rules. However, the user must use great caution if he is to successfully fudge a system as complex as SORT. Pitfalls will be many. For example, if a file has the format PPPPSPSPS..., and a sort is done using the 'S' option, the output file will probably not contain the first three primary records at all. This case occurs because when sorting using the 'S' option, pointers are generated for only the secondary records, prefixed by a pointer to the record preceeding the first secondary record of a record block. Since no secondary pointers were ever generated for the first three primary records, they are simply lost. It should be easy for the user to imagine what would happen to a file if a tertiary sort were attempted.

34.4.10 Key file drive number

There are three file systems associated with a sort. The first is, of course, the input file. The second is the output file. The third is the keyfile system. (The user only uses the output file - the keyfile system is a scratch file used by the system during sorting). There are actually two files which get opened during the sort for the keyfile system. They are *SORTKEY/SYS and *SORTMRG/SYS. These two files can grow to considerable sizes during the sorting procedure since they are proportional to the number of records and the size of the key field.

There are two considerations for the location of the keyfile system. The first is the problem of room. The keyfile must be on a drive with sufficient room to hold it. The second is speed. The greatest increase in speed occurs in removing the keyfile system from the same drive as the input file. Greater speeds can occur if it is, as well, not on the same drive as the output file.

Normally the SORT does a pretty good job of determining the best location of the two keyfile files and it shouldn't be necessary to specify anything for this. However, under complex circumstances, it may be desirable for the operator to specify the drive number for the keyfile. Should this be the case, the user should type in the <:DRk> specification as indicated in the general command format in Section 3.1.

34.4.11 Disk space requirements

A formula for determining the room in physical disk records that will be required for the SORT work files is:

$$R = \frac{NT(L+P+3)}{S} + 4T$$

where: R = Room in physical disk records required on disk.
N = Number of logical records in input file for which keys will be generated:
= number of records in file if not sorting on 'P' or 'S'.
= number of primary records in file if sorting on 'P'.
= number of secondary records in file if sorting on 'S'.
L = Length of the sort key in bytes.
P = 3 if sorting on secondary records,
0 if not sorting on secondary records.
T = number of sort key trains.
S = bytes per block of physical space available to the user (nominally 253 bytes)

The value of T can be computed exactly, but it is easier to make the general statement that short files will generate only one sort key train and longer files will generate more than one sort key train. Experience will soon develop empirical and intuitive knowledge for T evaluation for the user.

34.4.12 LINK into SORT from programs

There are three ways in which a SORT can be initiated:

1. From the keyboard via the DOS COMMAND HANDLER;
2. By using the DOS CHAIN command;
3. By loading and linking to SORT/CMD from an assembly language program.

Datashare users can invoke SORT by using the rollout

facility to start or continue a chain (see CHAIN and the DATASHARE User's Guide for more details).

Sort reserves for the user a nominal amount of storage normally occupied by the DOS DEBUG$ routine. The specific memory locations saved are 06144 through 06377. This permits the user to partially overlay his program with the SORT utility and regain control at the completion of the sort. Additionally, the next page of storage, 06400-06777, is available to the user if full image output records are to be generated. The DOS interrupt handler is disabled during the sort but is re-enabled upon completion of the sort. Of course, if the user has a foreground process running before and after the sort, the process must be controlled from within the memory not used by SORT, or when foreground is re-enabled it will vector to whatever SORT left in memory.

The steps to call SORT from an assembler program are as follows:

1. Close files 1, 2, and 3 if open.
2. Set MCR$ (01400-01543) with the command string terminated by a 015.
3. Load the SORT utility.
4. PUSH the stack.
5. Point HL to a parameter table with the format:

```
PTABLE  DA    LIMSTG
        DA    HEDING
        DA    EXITAD
```

6. RETURN

Where:

LIMSTG = the LIMITED OUTPUT SPECIFICATION string, terminated by a 015. If there is to be no limitation output specification, put 0. If there is a LIMSTG, it must exist entirely within the range 06144-06377. The LIMSTG must be exactly the characters as they would be entered from the keyboard. Examples follow.

HEDING = the HARDCOPY HEADING string, terminated by a 015. If there is to be no hardcopy output, put 0. If there is a hardcopy heading string, it must exist entirely within the range 06144-06377. The HEDING must be exactly the characters as they would be entered from the keyboard. Examples follow.

EXITAD = the first memory location to be executed upon successful completion of the sort. If the sort is to return to the

DOS upon completion, put 0. If there is a specific exit address, it must exist within the range 06144-06377. Normally, the instructions at the exit address will load and run the program to be run after the sort, or will re-load a control program of the user's own control system.

A simple example of loading and running sort from an assembler program would be:

```
                .
                .
 1.SRTCMD  DC    'SORT INFILE,OUTFILE',015  SORT COMMAND STRING
 2.SRTNAM  DC    'SORT    CMD'             NAME OF SORT UTILITY ON DISK
 3.PTABLE  DA    0                         NO LIMITATION STRING
 4.        DA    0                         NO HARDCOPY HEADING
 5.        DA    0                         NO SPECIAL EXIT ADDRESS
                .
                .
                .
 6.RUNSRT  LC    SRTNAM-SRTCMD             MOVE THE SORT COMMAND STRING
 7.        DE    MCR$                      TO MCR$
 8.        HL    SRTCMD
 9.        CALL  BLKTFR
10.        LC    -1                        LOAD THE SORT UTILITY
11.        DE    SORTNAM
12.        CALL  LOAD$
13.        PUSH                            PUSH THE SORT STARTING ADDRESS
14.        HL    PTABLE                    POINT TO THE PARAMETER
15.        RET                             RUN SORT
                .
                .
```

The above sequence of instructions could be located anywhere in memory, <u>except</u> lines 13 thru 15 must obviously reside in a portion of memory from 06144 thru 06377 to avoid being overlayed when the SORT utility is loaded from disk. The above instructions exemplify the simplest possible case of linking to SORT, in that only the SORT command and an INPUT FILE and an OUTPUT FILE are specified, all other options are defaulted. The above instructions have the same effect as calling SORT by entering the line:

```
SORT INFILE,OUTFILE
```

to the DOS COMMAND HANDLER.

Here is a line-by-line explanation of the instructions:

Line 1 defines the SORT COMMAND STRING. This is accomplished by a simple DC statement of a quoted ASCII string followed by a 015. The quoted ASCII characters are exactly the same that would be keyed in to the DOS COMMAND HANDLER if the sort were being initiated from the keyboard. The 015 is the string delimiter and is the same character that is placed after a string by the KEYIN$ routine when the ENTER key is depressed. The SORT command string can be up to 100 characters long <u>including</u> the 015 because the MCR$ area is 100 bytes long. Note that this is nineteen characters more than can be specified from the keyboard.

Line 2 defines the name of the SORT utility main overlay. Notice that the complete name of the SORT given here must be exactly the name as listed in the DOS DIRECTORY of files. The eleven ASCII characters in a file name specification include an eight character FILENAME and a three character EXTENSION. Since the FILENAME of SORT is only four characters, it must be followed by four spaces before the EXTENSION of CMD can be given.

Line 3 defines the beginning of the six-byte PARAMETER TABLE. The first two bytes of the parameter table specify the address of the beginning of the LIMITED OUTPUT SPECIFICATION string. In this example there is to be no limited output specification string specified, so an address of 0 is given.

Line 4 defines the address of the beginning of the HARDCOPY HEADING string. In this example there is to be no hardcopy output, so an address of 0 is given.

Line 5 defines the address of the EXIT ADDRESS, or the address to which the SORT is to exit when it is successfully completed. (If something goes wrong during the sort, exit is to the DOS.) In this example there is to be no special exit address, so an addess of 0 is given.

Line 6 begins the actual process of calling SORT from the program. Lines 6 thru 9 move the SRTCMD string from wherever it is in memory to the MCR$ area.

Line 10 specifies that SORT is to be loaded from wherever it is found in the disk drives that are on-line to the system. Refer to the DOS SYSTEM MANUAL if you are not familiar with the DOS LOAD$ routine.

Line 11 points to the name of the SORT utility main overlay in memory, given in SRTNAM, line 2.

Line 12 calls the DOS LOAD$ routine which finds the SORT main

overlay program on disk and loads it into memory, leaving the starting address in HL.

Line 13 puts the starting address of SORT on the P-counter Stack.

Line 14 points to the PARAMETER TABLE, lines 3, 4, and 5. The way that SORT knows that it is being run by the DOS COMMAND HANDLER or by a user program is by comparing the values of the HL contents and the top entry of the P-counter stack. If the values are equal, as they are immediately following a LOAD$, then SORT asks for a LIMITED OUTPUT SPECIFICATION string and a HARDCOPY HEADING string if they are specified in the SORT COMMAND string. If the values are not equal, then SORT checks the memory pointed by HL for the location of the LIMITED OUTPUT SPECIFICATION string, the HARDCOPY HEADING string, and an EXIT ADDRESS.

Line 15 effects the actual transfer of execution to the SORT utility. Since the starting address of the SORT was PUSHed onto the P-counter stack, a RETurn instruction JuMPs to the SORT starting address.

A DATASHARE program can link to SORT by executing a ROLLOUT instruction to a user-built CHAIN file which includes the SORT COMMAND LINE and, if specified, the LIMITED OUTPUT specification line and a HARDCOPY HEADING line, followed by the DSDBACK program to re-load the DATASHARE.

34.5 The use of CHAIN with SORT

The reader should first familiarize himself with CHAIN by thoroughly reading the CHAIN Section.

CHAIN is a system whereby the operator of a Datapoint Disc Operating System may pre-define a procedure sequence of his own programs, system commands and utilities (including keyboard answers to questions requested by these programs) and have them called and sequentially executed by a single name. This is especially powerful when using SORT since there may be a repetitive sequence of routines with complex parameterizations which would make good use of a simplification.

34.5.1 How to set up a chain file for SORT

The author of a chain file only needs to remember that ALL questions that the system requests INCLUDING those initiated by the executing programs MUST BE ANSWERED from the chain file just as though they would be typed in from the keyboard.

For instance, the initiation of a sort 'SORT INFILE,OUTFILE;I3-42' could be done through chain. To do this, use the Editor to type in that exact sequence of characters into a file. Note that the file will, in this case, consist of a single line as typed above. The file can be any name, but for purposes of simplifying the explanation, it shall be referred to as CHAINFIL. If CHAINFIL consists of that single line, and if the operator types the command 'CHAIN CHAINFIL' to the DOS, the SORT specified above would be initiated. If the 'L' specification were included in the statement above, then SORT would ask for another line of information. In this case, the file CHAINFIL would have to have two lines in it with the first being the SORT command and the second being the limited output file format specification.

34.5.2 Naming a repetitive SORT procedure

Frequently there are sorts and printouts and other procedures which occur together and for which a name invoking the procedure would be a great simplification.

For instance, in the telephone directory example above, the process of sorting the file into a limited output file and then listing it on a local printer could be procedurized as follows:

```
SORT EMPFILE,TELFILE;L80-110
80-110,'  - ',171-180
LIST TELFILE;XL
TELEPHONE DIRECTORY FOR XXXXXXXXXX CORPORATION
```

Note that there are four statements. The first is the SORT command. The second is the answer to the limited format initiated by the 'L' in the SORT command. The third is the DOS LIST command with the specifiers of 'X' which says 'without line numbers' and the 'L' which, here, means local printer. Then there is a forth line which the LIST command requests - the heading. This question must also be answered in the chain file. If the above four statements were placed in a file by the Editor (or by any other means, for that matter) and then CHAIN were invoked with that file specified, the result would be a sorted telephone directory from the personnel files appearing on the printer.

34.5.3 Initiating a SORT from another program

The chain file (CHAINFIL above) could have been created by any Datapoint system which can write a file. This makes the concept even more powerful since programs can create or modify subsequent procedures of itself, other programs, system commands and utilities. RPG II especially can make good use of this.

34.5.4 Using CHAIN to cause a merge

Consider a situation wherein a system has a master file called 'MASTER' and a file of records to be added, in sequence, to the master file called 'ADDFILE'. To merge these two files in sorted sequence at the end of each day would normally require a sequence of keyed in operations which are somewhat complicated and error prone. CHAIN can cause an effective MERGE and assign it a single name as follows:

```
SAPP MASTER,ADDFILE,MASTER
SORT MASTER,SCRATCH;1-20
KILL MASTER/TXT
NAME SCRATCH/TXT,MASTER/TXT
```

Note that the procedure:
1) appends the ADDFILE to the MASTER file.
2) Sorts the extended MASTER file into a SCRATCH file.
3&4) Renames the SCRATCH file as the new MASTER file. Thus, it is apparent that a merge can be effectively achieved using SORT and by using chain to pre-define the procedure.

34.6 SORT Execution-Time Messages

This subsection describes the operator messages that SORT may display on the CRT screen during execution. Some of the messages are monitor messages to keep the operator informed of the progress of the program, while other messages are error messages.

DOS.X Ver.n SORT - DATE

This message is the SORT sign-on.

SORT OVERLAY MISSING.

This message is displayed if the SORT/OV1 file is not on the same drive as the SORT/CMD file.

INPUT FILE REQUIRED.

This message is displayed if no filename was specified for the first file specification. This would happen if a command lines such as:

SORT ,OUTFILE or SORT /TXT,OUTFILE

were entered.

OUTPUT FILE REQUIRED.

This message is displayed if no filename was specified for the second file specification AND if the 'L' and 'H' options were not specified.

BAD DEVICE SPECIFICATION.

This message is displayed if a drive specification in a file specification was not entered in exactly the format; DRn where n is a valid drive number.

OUTPUT FILE SAME AS INPUT.

This message is displayed if the FILENAME and EXTENSION of the INPUT file and the OUTPUT file are the same, and the DRIVE NUMBER for each file is the same or not specified for EACH file.

INPUT FILE NOT FOUND.

This message is displayed if the INPUT file could not be found on any drive on-line to the system if no drive was specified, or on the drive given if a drive was specified. If no extension is supplied in the file specification an extension of TXT will be assumed; in this case if a file FILENAME/TXT is not on-line or on the drive specified then the INPUT file will not be found.

INPUT FILE RIB ERROR.

This message is displayed if a read parity error occurs when the INPUT file's RIB is checked to determine the INPUT file's length.

KEY FILE SPECIFICATION ERROR.

This message is displayed if a FILENAME or EXTENSION is given for the KEY DRIVE specification.

KEY FILE DEVICE SPECIFICATION ERROR.

This message is displayed if the drive specification for the KEY file was not exactly in the format: DR# where # is a valid drive number.

SORT KEY FILE PLACED ON DRIVE #

This message is displayed if the KEY DRIVE was not specified on a multi-drive system. The message is to notify the operator of the location of the KEY file. The # stands for a valid drive number.

OPTION FIELD ERROR.

This message is displayed if a semicolon ; is entered at the end of the SORT command line but is not followed by any option specifications.

OPTION SPECIFICATION DUPLICATION.

This message is displayed if a command line such as:

```
SORT INFILE,OUTFILE;DLA
```

were entered. The 'D' and 'A' options are both variations of the ORDER option, and obviously both cannot occur simultaneously.

HARDCOPY ONLY IF LIMITED OUTPUT SPECIFIED.

This message is displayed if the 'H' option is specified but the 'L' option was not given previously.

ILLEGAL HEADER SPECIFICATION.

This message is displayed if the 'P' or 'S' option is given but is immediately followed by the 015 byte -- the ENTER key.

ILLEGAL HEADER KEY EVALUATION.

This message is displayed if the character immediately following the 'P' or 'S' option is not '=' or '#'.

ILLEGAL SORT KEY SPECIFICATION.

This message is displayed if a key position of 0 or greater than 249 was specified, or if a key position was not terminated by , or - or 015, or if a two-position key was not terminated by , or 015.

SORT KEY TOO LONG.

This message is displayed if the total sort key is longer than 100 characters long.

OVERLAPPING SORT KEY SPECIFICATIONS---SORT OPTIMIZED.

This message is displayed if the same record positions were specified for more than one sort key group. SORT does not repeat duplicate positions in sort key generation and thus saves processing and disk read/write time.

OVERLAPPING SORT AND HEADER KEYS---SORT OPTIMIZED.

This message is displayed if the same record position is specified as a sort key position and a header indication position. The position is removed as a sort key position and the key is thus shortened. The effect is as for the previous message.

LIMITED OUTPUT FILE FORMAT:

This message is displayed if SORT has accepted the SORT command line including all option specifications and if the 'L' option has been given. The operator must enter the limited output specification line.

NULL LIMITATION SPECIFICATION.

This message is displayed if the 'L' option was given but the limitation specification was only 015 -- the ENTER key. If the

'L' option is given then a non-empty limited output specification string must also be given.

INVALID LIMITATION SPECIFICATION.

This message is displayed if the limited output specification does not fit the syntax given in subsection 28.3.6 of the SORT Section. Usually the fault is that a comma was not placed between option specification groups, or double quotes " were used instead of single quotes ' .

ENTER THE HARDCOPY HEADING:

This message is displayed when the limited output specification has been accepted and if the 'H' option was given. The operator must enter from 0 to 79 characters of information which will be printed at the top of each page printed during SORT output generation.

SEQUENCE FILE NAME REQUIRED

This message is displayed when the sequence file field is blank and the file specification fields have not been terminated with a semi-colon or an end of line designator.

SEQUENCE FILE NOT FOUND

This message is displayed when SORT requests the sequence file be OPENed and DOS cannot locate the file on the disk drive indicated. Note that if the drive is not specified, the drive on which the SORT/CMD resides is implied.

SEQUENCE FILE FORMAT ERROR A

This message is displayed when SORT determines that the sequence file specified is not an absolute object file.

SEQUENCE FILE FORMAT ERROR n

This message is displayed when SORT receives an error return from LOADX$ when an attempt is made to load the sequence file. The value of n may be 0-6 and is defined as follows:

0 If file does not exist
1 If disk drive is off-line
2 If directory parity error

3 If RIB parity fault
4 If file parity fault
5 If off end of physical file
6 If record of illegal format

LIMITATION SPECIFICATION OVERFLOW

This message indicates that limited output parameters entered require more memory (256 bytes) than allocated by SORT.

INTERNAL ERROR -- GET SYSTEM HELP !!!

This message indicates a probable hardware error occurred during a limited output string sort. SORT cannot continue executing.

MERGE FILE OVERFLOW

This message indicates not enough disk space is available for the merge file.

FULL IMAGE OUTPUT RECORDS

This is an informative message to the operator that full image records are being output by SORT.

OUTPUT FILE OVERFLOW

This message indicates not enough disk space is available for the output file.

THE FOLLOWING MESSAGES MAY BE DISPLAYED DURING SORT INITIALIZATION IF SORT WERE LINKED TO BY AN ASSEMBLY LANGUAGE PROGRAM:

INVALID LIMITATION STRING ADDRESS.

INVALID HARDCOPY HEADING STRING ADDRESS.

INVALID USER EXIT ADDRESS.

One of these messages is displayed if the corresponding entry in the parameter table linkage data was not either 0 or in the range 06144-06377 inclusive.

LFT ENTRIES 1->3 NOT CLOSED WHEN SORT ENTERED.

This message is displayed if the user left one of the logical files 1, 2, or 3 open upon linking to the SORT utility.

LIMITATION STRING MISSING.

This message is displayed if the 'L' option was given in the SORT command string but the pointer to the limited output format string in the parameter table linkage data was 0, indicating no limited output format string specified.

HARDCOPY HEADING STRING MISSING.

This message is displayed if the 'H' option was given in the SORT command string but the pointer to the hardcopy heading string in the parameter table linkage data was 0, indicating no hardcopy heading string specified.

THE FOLLOWING MESSAGES ARE DISPLAYED AFTER THE SORT INITIALIZATION IS COMPLETED:

BUILDING SORT KEY TRAIN n.

This message is displayed when all parameter specifications have been accepted and SORT has started the extraction of the sort keys from records of the INPUT file and is writing them to the *SORTKEY/SYS file.

SORT KEY FILE OVERFLOW.

This message is displayed if there was not adequate room on the KEY DRIVE to hold the *SORTKEY/SYS file. If *SORTKEY/SYS file overflow occurs the file is deleted from the disk before the message is displayed.

NULL OUTPUT FILE.

This message is displayed if no sort key records were generated. If no sort key records are generated SORT cannot re-order the INPUT file, thus no output generation would be useful.

INTERMEDIATE SORT PASS n.

This message is generated during sorting of the sort key trains on the *SORTKEY/SYS file. The only actual sorting done during a sort is that which can be done on the initial sort key trains, which are made short enough that they will fit in memory. After the sorting of the keys within each initial train, the trains are merged sixteen abreast into larger trains, repeatedly until only one train remains.

INTERMEDIATE MERGE PASS n, TRAIN n.

This message is displayed if more than sixteen sort key trains exist during a merge pass. The intermediate merge pass number is the Nth iteration of the merge process. The train number is the number of the train being output by the merge pass. If more than one train is output by an intermediate merge pass then at least one more intermediate merge pass will be required. If more than sixteen trains are output by an intermediate merge pass then at least two more intermediate merge passes will be required, and so on.

FINAL MERGE: SORT TRAIN n.

This message is displayed during the generation of the output file from the data in the now fully sorted and merged sort key file and from the records in the INPUT file. The sort train number corresponds to the current state of progress as measured against the number of trains generated by the next to the last intermediate merge pass.

CHAPTER 35. SUR COMMAND

35.1 Purpose

When a specific disk is used for more than one purpose, some inconveniences occasionally turn up. Assume for a moment that a user has a disk which he is using for program generation on each of two more or less unrelated projects. When he uses the CAT command, for instance, he will normally see a whole range of files, some of which are not related to the project he may be currently interested in. Or, he may begin editing a new file on the disk, only to find that another user of the same disk may have already had a file of that name. At times like this, it would be convenient to logically partition the directory so that a user would only have a portion of it, the portion he is currently interested in, available to him at one time.

A more concrete example is the DOS itself and its various commands. Obviously Datapoint's DOS.A, DOS.B, and DOS.C bear a strong resemblance to each other. The DOS and most of the command files are configured at assembly time through conditional assembly and equates to support a given disk controller and specific file structure. The result is several different object code files, all with a /ABS extension, for each single source file with a /TXT extension. Yet it is desirable for a number of reasons to keep all of the object code files for all the DOS and commands on a single drive.

Without the DOS subdirectory facility, it is not permitted to have two files on a given logical drive with the same name.

35.2 About Subdirectories

The use of the SUR (Subdirectory Utility Routine) command allows the user to logically partition the directory on a given disk into several smaller subdirectories. Each such subdirectory can then contain zero or more files, up to the maximum number of 256 files per logical drive. Each subdirectory on a disk has a unique name. Two subdirectories always exist on all drives; these are called SYSTEM and MAIN. The names for the other subdirectories are assigned by the user as he establishes them, and follow the same rules as for any standard DOS file name. As a subdirectory is created, the name specified by the user is related

to a unique number which is referred to as the subdirectory number. The relationship between subdirectory names and subdirectory numbers is not unlike the relationship between DOS file names and physical file numbers. A given subdirectory may have different numbers on different drives, even though the subdirectory name is the same.

It is important to realize that subdirectories are not a way of getting more than 256 files on a drive. This they cannot do. The thing that subdirectories are good for is partitioning the directory and restricting the scope of a file name. This allows several files of the same name to exist on one disk at the same time, without causing the DOS to become confused as to which is the one to be referenced at any time. The way the DOS achieves this is that each of the files is in a "different subdirectory" from each other, and hence is uniquely identified even though the name and extension may be identical.

35.2.1 Creation of Subdirectories

Subdirectories are created with the SUR command. All that is required is to specify a name for the proposed subdirectory and request its creation. Creation of a subdirectory does not actually result in any real change to the directory on disk at all; all it does is to cause the specified name to be entered into a table, kept on disk, which relates each subdirectory name with its subdirectory number. The user is allowed to specify which drive he wishes to create the subdirectory on; if he does not indicate a specific drive, the named subdirectory is placed onto all on-line drives if possible.

35.2.2 Deletion of Subdirectories

Subdirectories are deleted with the SUR command. The user specifies the name of the subdirectory he wishes to remove and requests its deletion. Deletion of a subdirectory does not result in KILLing the files within the range of that subdirectory. If a subdirectory to be deleted contains one or more files, the files are first moved from that subdirectory to the one called MAIN before the named subdirectory is deleted. The user is allowed to specify from which drives the subdirectory is to be deleted; if he does not indicate a specific drive, the named subdirectory is deleted from all on-line drives on which it appears.

35.2.3 Being "in a Subdirectory"

The user can define at any time which of the subdirectories on each of his disks contain the current files he is interested in. This is done with the SUR command by specifying the name of the subdirectory containing the files of current interest. This action causes him to be placed "into" the named subdirectory on the drive specified. (If no specific drive is mentioned, he will be placed "into" the subdirectory specified on all on-line drives containing a subdirectory with the given name). It is appropriate to point out that the current subdirectory on each drive need not have the same name; for example, the user could easily be in subdirectory PROGRAMS on drive zero and in subdirectory DATABASE on drive one at the same time.

Once in a specific subdirectory on a drive, that state does not normally change until the user requests being placed into a different subdirectory (again via the SUR command) or re-boots the DOS. Rebooting the DOS causes the user to be placed into the subdirectory named SYSTEM on all drives.

35.2.4 Scope of a File Name

When a program accesses a file under DOS, it tells DOS the name and extension of the file it is looking for and either indicates one specific drive which the DOS is to search for the file, or requests that the DOS look on all on-line drives. In order for the DOS to "find" the given file, the DOS must find a file whose name and extension exactly match the ones specified by the requesting program. If no such file can be found, the DOS returns indicating that the specified file cannot be found and therefore probably does not exist.

When subdirectories are in use, this matching of name and extension is expanded so that in addition to a file's name and extension matching those specified by the requesting program, the file must also be within either the current subdirectory (for that drive) or the one called SYSTEM in order to be "found".

Therefore the scope of a file name can be more or less defined via the following: when a user is in subdirectory X on drive Y, files can be "seen" by his program only if they are in either subdirectory X or subdirectory SYSTEM. Files in any other subdirectory will not appear to exist.

35.2.5 About Subdirectory SYSTEM

It has been shown that files in the subdirectory named SYSTEM are special in that they can be accessed regardless of which subdirectory the user is "in" on a specific drive. Likewise, a special situation also occurs when the user is "in" the subdirectory named SYSTEM. When the subdirectory named SYSTEM is the current subdirectory on a given drive, all files on that drive are accessible regardless of which subdirectory they themselves are actually in.

A little caution must be used when a user is in subdirectory SYSTEM on a disk with multiple files of the same name and extension. The caution is that, although each of the files is still associated with one and only one subdirectory, all of the files on a disk are available when the user is "in" the SYSTEM subdirectory. The result is that in this situation, one of the files of the desired name and extension will be referenced; which one is referenced is, however, undefined. Therefore, good practice dictates that if a user has more than one file with the same name and extension on some drive, that he make a point of always knowing which subdirectory he is in (and that it is <u>not</u> SYSTEM) if it matters to him which of his files he references.

35.2.6 Files vs. the User Being "in a Subdirectory"

It is important not to confuse the two distinct concepts of a <u>file</u> being in a subdirectory as opposed to that of [a user] "being in a subdirectory".

A <u>file</u> being in a specific subdirectory is a way of saying that the file cannot be accessed when the current subdirectory is neither that specific subdirectory nor SYSTEM. This relationship, that of a file being in a specific subdirectory, is retained more or less permanently; if a file is placed in subdirectory SUBDIR1 today on a disk, the disk can be removed and stored on a shelf; if tomorrow the disk is taken down from the shelf and re-mounted, that file will still be in subdirectory SUBDIR1.

A <u>user</u> being in a specific subdirectory is a way of saying that the subdirectory in question is "the current subdirectory" on one or more logical drives. The "current subdirectory" on a drive is less permanent and reflects the use of the SUR command since the previous time the DOS was bootstrapped.

As in most computer-related things, the best understanding of subdirectories is attained through experimentation.

35.2.7 Getting a File into a Subdirectory

In general, there are three ways to get a file into a given subdirectory. The easiest and probably most common of these is automatic. Whenever a file is created, it is always placed into the current subdirectory on the drive on which it is created.

Once a file has been thus created, it can be moved between subdirectories with the NAME command. The NAME command can take a file within the scope of the current subdirectory and put it into the current subdirectory if it is not already (which is useful if either the source or destination subdirectory is SYSTEM) or can place it into any other subdirectory the user might wish to put it into.

35.3 Usage

The SUR command is parameterized as follows:

SUR [<name>][/<function>][:DR<n>][,<new name>]

The function performed by SUR is determined by the absence or value of the <function> field and the name field, as described below.

35.3.1 Establishing a "Current Subdirectory"

If the function field is not given, SUR establishes the named subdirectory as the current subdirectory on all drives on which the named subdirectory exists. If the named subdirectory does not exist on one or more drives, the current subdirectory on any such drives is unaffected. If a specific drive is mentioned, then only the current subdirectory on the specified drive is subject to change.

35.3.2 Creating a Subdirectory

If the function field is /NEW, SUR creates the named subdirectory on all drives on which the named subdirectory does not exist. The current subdirectory is not affected by the operation. If a specific drive is mentioned, then the named subdirectory is only created on the specified drive.

35.3.3 Deleting a Subdirectory

If the function field is /DEL, SUR deletes the named subdirectory on any drives on which the named subdirectory exists. If any files are in the named subdirectory, they are moved to subdirectory MAIN before the named subdirectory is deleted. If the subdirectory being deleted is the current subdirectory on that drive, the current subdirectory is also changed to MAIN. Subdirectories SYSTEM and MAIN cannot be deleted. If a specific drive is mentioned, then the named subdirectory is only deleted from the specified drive.

35.3.4 Renaming a Subdirectory

If the function field is /REN, SUR renames the named subdirectory on any drives on which the named subdirectory exists, to the name specified in the new subdirectory name field. If any files are in the named subdirectory, they will be in the subdirectory specified by the new subdirectory name field upon completion of the operation. Subdirectories SYSTEM and MAIN cannot be renamed. If a specific drive is mentioned, then the name of the named subdirectory is changed only on that specified drive.

35.3.5 Displaying Subdirectories

If the subdirectory name field is not given, SUR displays the names of all subdirectories on all on-line drives. The format of the listing is similar to that provided for file names by the CAT command. The number in parentheses to the right of each subdirectory name is the subdirectory number associated with that name (in octal); an asterisk indicates the current subdirectory on each drive. If a specific drive is mentioned, then only the subdirectories present on the specified drive are displayed.

CHAPTER 36. ADVANCED PROGRAMMER'S GUIDE

36.1 General Background Information

The object of an operating system is to allow maximal use of the capabilities of a computer with minimal effort. A Datapoint 1100, 2200 Version 2, or 5500 computer with a Datapoint disk memory unit attached is capable of a very sophisticated mass information storage structure and a multitask environment. The sophistication of the mass storage structure allows efficient use of the available space while maintaining operator convenience and error recovery. The multitask environment allows the execution of several functions simultaneously. With the preceding in mind, a complete set of system routines and operator commands are provided.

36.2 Operator Commands

The operating system contains a routine that interprets user commands given at the keyboard and performs the tasks indicated. A large set of commands are supplied with the system which provide the user with facilities for creation, modification, and execution of files, along with a dynamic debugging facility. These include a general purpose editor and many useful disk file handling commands. A complete set of CTOS compatible cassette handling commands are also provided, allowing the user to transfer files between the disk and cassettes.

Since the commands are actually programs which the system loads and executes to perform the task required, the command language is naturally extensible to include any program the user may desire, thus leading to a powerful keyboard facility.

36.3 System Structure

The operating system proper resides within the first 8K of memory. Of this, only the first 2.8K is necessary for the support of the disk. The debugging tool and CTOS compatible keyboard and display routines occupy through 4K, the cassette handler through 5.4K and the command handler through the rest. When the system is bootstrapped from the rear cassette, the first 768 bytes are loaded with a loader, entry point table, and interrupt handler. The small size of the disk file handling routines is due to the

use of overlays for the file opening and closing functions. An overlay is also used to contain most of the system error messages, allowing fully descriptive messages without using a prohibitive amount of main memory.

The operating system supports one disk controller with one or more physical disk drives attached. Each physical drive contains one disk unit which is considered to be one or more logical drives, each of which consists of a completely self-contained information structure. Each logical disk can contain up to 256 files, the maximum length of any one of which being determined by the particular DOS in use; but in no case may the size of any file exceed the capacity of the logical drive on which it resides. File space is allocated dynamically while maintaining as much physical contiguity as possible, thus enhancing access time and storage efficiency.

36.4 Interrupt Handling

A set of routines is loaded by the bootstrap that allows the user to make effective use of the interrupt facility in the Datapoint computers. These routines schedule the execution of interrupt driven processes, provide facilities for these processes to be turned on and off, and provide a mechanism via which these processes can be made to execute in a convenient manner. Note that since these routines are loaded only by the bootstrap action, interrupt driven processes are not stopped by the loading of the system or other programs. The DOS cassette handling routines make use of this interrupt facility to allow slewed reads and writes when moving data between the disk and cassettes, greatly increasing the rate of transfer.

36.5 System Routines

Routines within the operating system provide the programmer with facilities for dealing with the disk, cassettes, keyboard, and display. Each category of routine has an entry point table to allow system changes without necessitating a change in the user's code. Since each category has its own table, those routines not needed may be overlayed by the user. The keyboard and display routines are identical in parameterization and function to the CTOS routines. The cassette routines perform the same functions as the CTOS routines but are parameterized differently. All of the routines execute with interrupts enabled and have a full set of error traps, enabling the user to deal with all errors except those fatal to the system.

36.6 Physical Configuration Requirements

The minimal physical configuration required to support the disk operating system is a Datapoint 1100, 2200 or 5500 computer (minimum 16K memory) and a Datapoint Corporation disk memory peripheral. (Note that the Version 1 2200 is not capable of running the DOS). As mentioned earlier, users with Datapoint 5500 computers and wishing to use the full 5500 disk operating system will need more than 16K, with the full complement of 48K user memory recommended.

The choice of computer (1100, 2200, or 5500) and disk drive type (flexible diskette, cartridge disk, or full eleven-high disk pack drive) will determine which versions of DOS a user may choose from.

36.7 Program Compatibility with Different DOS

The various versions of Datapoint DOS vary somewhat in internal disk structuring and in small related details. (Such detailed information is provided in the DOS System Manual corresponding to the user's individual DOS). In general, if a programmer uses the information contained in this User's Guide in writing his program, that program should run under any of the Datapoint Corporation DOS, without modification.

Use of information contained within the DOS System Manual with regard to internal disk structuring details, absolute physical locations of system tables, assumptions regarding the format and contents of internal physical disk addresses, and the like, should be avoided within user programs as it will tend to impair compatibility with different DOS and obstruct ease of future upgrading to higher capacity, more cost effective disks and more powerful processors.

CHAPTER 37. OPERATOR COMMANDS

Files are identified from the console by a NAME, EXTENSION, and LOGICAL DRIVE NUMBER. The NAME must start with a letter and may be followed by up to seven alphanumeric characters. For many commands, this is the only information that must be supplied. The EXTENSION must start with a letter and may be followed by up to two alphanumeric characters. It further defines the file, usually indicating the type of information contained therein. For example, TXT usually implies user data files or source information (e.g. DATASHARE, ASM, DOS DATABUS, or SCRIBE source lines), ABS usually implies program object code records that can be loaded by the system loader, and CMD usually implies programs that implement commands given the DOS from the keyboard. Most commands have default assumptions concerning the extensions of the file names supplied to them as parameters. However, extensions may otherwise be considered as an additional part of the name. The LOGICAL DRIVE NUMBER specifies which logical drive is to be used. It is given in the form DR(n) or D(n), where (n) is zero through the maximum number of drives supported within the user's configuration and the specific DOS he is using. If the drive is not specified, the system searches all drives starting with zero. Note that each logical drive contains its own directory structure. Specifying the drive number enables one to keep programs of the same NAME and EXTENSION on more than one drive.

Files are always created implicitly. That is, the operator never specifically instructs the system to create a given file. Certain commands create files from the names given as their parameters. Since space allocation is dynamic, the operator never specifies how many records his file contains.

Deleting files is made somewhat more difficult to protect the user from accidentally destroying valuable data. Files can be protected against deletion or both deletion and writing. In addition to this, the operator must always explicitly describe the file he is deleting and even then must answer a verification check stop before the actual deletion occurs.

The system has no explicit RUN command since, to execute his program, the user simply mentions its name as the first file specification on the command line. This is the mechanism via which both commands and user programs alike are executed. The first file specification may be followed by up to three more, depending upon the requirements for parameterization of the

program being run. A file specification is of the form:

```
NAME/EXTENSION:DRIVE
```

where any of the three items may be null (except the NAME must be given in the first specification which denotes the program to be run). Note that the / indicates that an extension follows and the : indicates that a device specification follows. If either of these items is not given, the corresponding denotation character is not used. For example:

```
NAME/ABS:DR0
NAME/ABS
NAME:D0003
NAME
```

are all syntactically correct. File specifications may be delimited by any non-alphanumeric that would not be confused with the extension and device indicators. For example:

```
COPY NAME/TXT,NAME/ABS
COPY NAME/TXT NAME/ABS
COPY NAME/TXT/NAME/ABS
```

will all perform the same function. If an extension is not supplied in the first file specification, it will be assumed to be CMD. In the above examples, COPY/CMD will be used for the complete file name sought in the directory for the command program name. Note that if one wanted to run a file he had created with extension ABS, he would simply enter

```
NAME/ABS
```

and his program would be loaded and executed. If the name given cannot be found in the directory or directories specified, the message

```
WHAT?
```

will be displayed. Note that the DOS can load any object code at or above location 01400 (octal).

CHAPTER 38. SYSTEM STRUCTURE

38.1 Disk Structure

A disk, whether a flexible diskette, cartridge, or pack, is a self contained information structure when used with the DOS. A disk's tables reference only information on the disk itself, and it is assumed that the structure of the disk will not be changed without these tables also being changed.

The smallest structural unit of information on the disk is called a cluster and is composed of some fixed number of 256-byte sectors. (The number of sectors per cluster varies with different DOS but in general is between three and thirty-two sectors per cluster.) Clusters do not span track or cylinder boundaries, and one track of the disk will generally contain one or more clusters. Since the cluster is the smallest allocatable unit of storage on the disk, one cluster represents the minimum possible file size.

A small portion of each logical disk is reserved for several special tables maintained by the system. These tables include the Cluster Allocation Table (CAT), Lockout CAT, and the Directory. These tables are maintained in duplicate for backup purposes. This helps to insure that a software or disk error will not result in possible massive loss of data.

The CAT is a bit map of the clusters that are not available for new space allocation. Each bit represents one cluster, and generally each byte represents one cylinder. Thus, the location of the byte in the table is equal to the number of the cylinder it is representing. Note that since not all 256 bytes are needed for this bit map (no supported disk has 256 cylinders), these excess entries are marked as not being available. The last byte of the CAT is the auto-execute PFN, which specifies the physical file number of the file to be automatically executed when the DOS is loaded. This number being zero implies that no file is to be automatically executed since physical file zero is the number of the DOS itself.

The Lockout CAT is similar to the CAT but is written only at the time of DOS generation of each disk and optionally during the processing of the REPAIR command (as discussed in the appropriate system guide). This sector is a copy of the CAT after the DOS

GENERATION program has certified the disk but before any files have been allocated on it. It therefore provides a sector indicating which areas of the disk have been flagged as bad during the DOS GENERATION certification process. This is used in conjunction with the main CAT to avoid allocating files onto known bad places on the disk.

The Directory consists of sixteen sectors. Each sector contains sixteen entries of sixteen bytes each. Each entry is associated with a number called the physical file number (PFN). This number is some function of the physical location of the entry in the directory (which function may vary for different DOS).

The DOS loader (resides between 0 and 01000 in memory) is parameterized by a logical drive number and a physical file number. It indexes the directory on the given disk by the physical file number according to the individual DOS's directory mapping function to obtain the file's physical starting location (part of the information within the directory). Physical files zero through seven are reserved for system usage. File zero contains all of the code for the DOS (except for the overlays) that resides above location 01400. Files one through six contain code for executing the following functions:

1 - PREP - create a new file
2 - CLOSE - close a file and delete if indicated
3 - OPEN - open an existing file
4 - ALLOC - allocate more space for a file
5 - ABORT - display an error message
6 - SCREEN - initialize the RAM display if present

These are overlays that reside in the area between 04000 and 05400. File seven is used for by the DOS subdirectory facility, described in the SUR command section, and for the DOS FUNCTION overlays.

The physical file number of a file is written in every record of that file for error control purposes. Otherwise, physical file numbers are used only to parameterize the system loader. At higher levels, files are parameterized by a symbolic name which is also contained within the directory entry. Other information contained within a directory entry is the physical disk location of the first cluster of the file, and protection bits to disable either deletion of the file or both deletion of and writing into the file.

Note that the name as stored in the directory consists of eleven bytes of data. The command interpreter, which handles

information given at the keyboard by the operator, deals with an eight byte name and a three byte extension (see the chapter on Operator Commands). This is, however, purely a convention of the command interpreter and has no significance in relation to the internal format of the directory. When system routines which deal with file names are used, eleven bytes are provided for the parameter, which is always dealt with as a monolithic item.

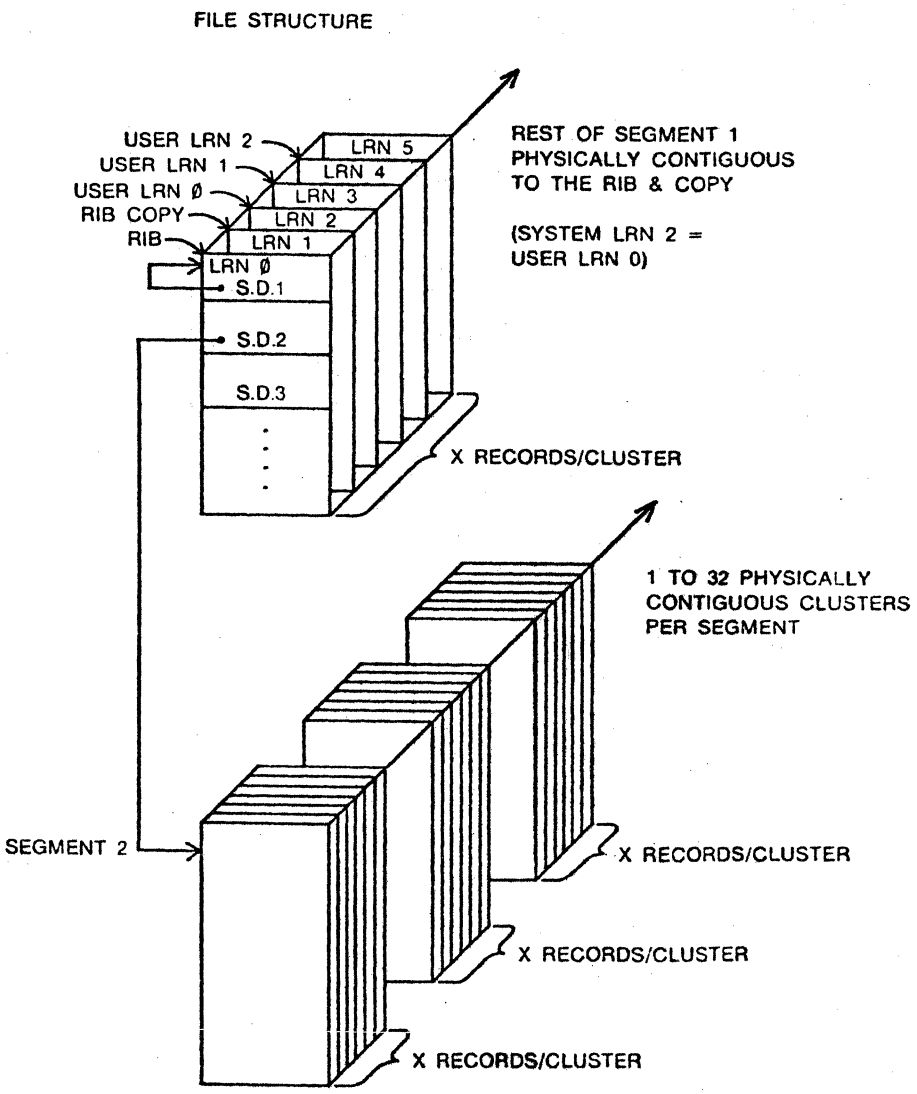
A file consists of logically contiguous records (there is a one to one correspondence between records and physical disk sectors). These records are allocated in one or more physically contiguous groups of clusters where each contiguous group is called a segment. This segmentation is employed to allow the dynamic allocation and de-allocation of disk space without having to move information contained in other files.

Internally, the DOS references records by a number, called the Logical Record Number (LRN), that starts at zero. The first two logical records (zero and one) contain a table, followed by its copy, that lists the location of each of the segments that make up the file. Logical records two through the last record in the file contain the user's data. These records are referenced by a number specified by the user that starts at zero. Therefore, system logical record two is user logical record zero.

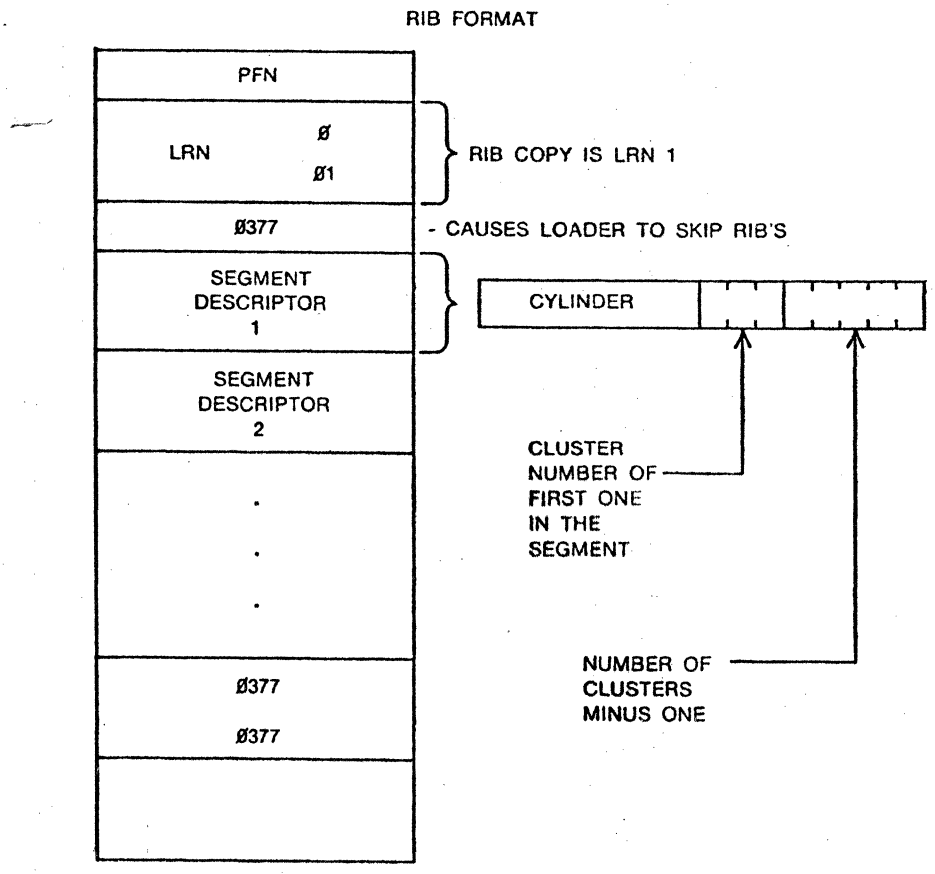
The table that describes the segments comprising the file is called the Retrieval Information Block (RIB). The working copy is kept in system logical record zero and its backup copy in record one. The backup copy is always written immediately after the working copy is written. It exists to allow recovery if the working copy shows a parity failure in later use. Since the 1100 and 2200 DOS always write with the write/verify mode of the disk controller, this situation should only occur if a power failure occurs while the working copy is actually being written on the disk (the actual writing of a sector on the disk surface only takes a millisecond or two, except for the flexible diskette). Since RIB updates occur infrequently, the probability of this kind of failure is extremely small. The CAT, Lockout CAT and Directory copies are treated in a manner similar to the treatment of the RIB copy for exactly the same reasons and exhibit the same small probability of requirement for backup. However, it is important that these backup facilities exist to prevent possible massive loss of the user's data.

The following figure depicts the file structure described above. Note that the logical record numbers indicated in the second figure are system numbers and that the LRN 2 shown is actually user LRN 0. The upper portion of the sketch shows the

first segment broken down into its individual records. The first record points to all of the segments in the file (including the segment that contains the first record itself). The lower portion of the sketch shows the next segment broken down into its individual clusters.



The RIB contains the physical file number in its first byte, as in all records within the system, and a logical record number in the next two bytes, also as in all records within the system (the LRN of the working copy is zero and of the back-up copy is one). The fourth byte of the RIB always contains an 0377. The rest of the 252 bytes within the RIB contains up to 126 segment pointers (called descriptors). The first segment descriptor points to the segment containing the RIB itself and always exists. If the list is shorter than 126 segments, a terminator consisting of two 0377's appears to denote that no descriptors follow. When the file is 126 segments long, the pair of 0377's does not exist. A single segment may be between 1 and 32 clusters long. This length is specified by the rightmost five bits of the second byte in the descriptor which is the length minus one. The maximum of 32 clusters per segment is due to the five length bits available in the RIB's segment descriptors. An additional limitation exists which further reduces the maximum number of clusters per segment for certain DOS, that restriction being that the total number of sectors per segment must be less than 256. The left three bits of the second byte, together with the first byte, specify the cluster where the segment begins. The following figure shows the format of the RIB:



38.2 Disk Data Formats

The DOS itself does not deal with the user's data below the record level. It only keeps track of where the records are, allowing the user to format the data in any manner he pleases. The user is presented with records that are 253 bytes long. The system keeps the physical file number in the first physical location of each sector and the system logical record number of the given record in the second (LSB) and third (MSB) physical locations of each sector. This is done to insure that the record obtained is the record desired. The last 253 bytes may contain anything the user chooses. There are, however, some assumptions made by the DOS and the programs supplied with it that deal with disk data. These assumptions fall into two classes: system loader object records and symbolic data records. The first class contains all records that are to be loaded into memory by the DOS loader. The second class contains all records that are to be handled by the standard data handling programs. These programs include the general purpose editor, the assembler, DATASHARE, RPG II, DOS BASIC, and the DATABUS programs (both source lines for the various compilers and data records handled by the resulting programs).

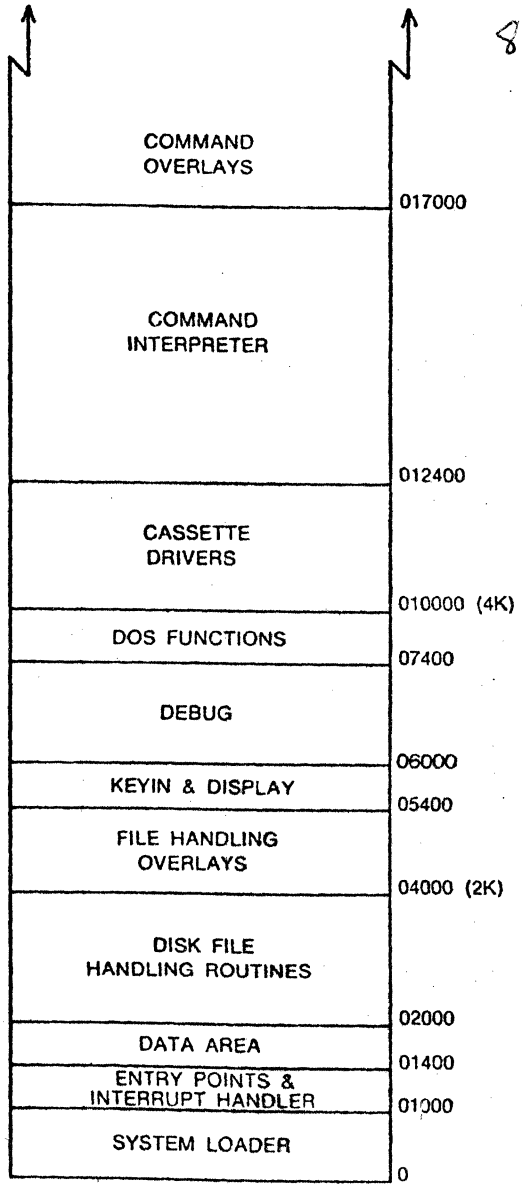
A record that is to be loaded by the system loader must have the following format:

Any number of the data blocks may appear in a record. The leading byte being 0 indicates a data block follows and 0377 indicates end of record. The special case of N being zero is used to indicate an end-of-file. In this case, the HL given is taken to be the starting address if one is to be used.

A record that is to be dealt with by one of the standard data handling programs must have the standard text file format described earlier in the chapter on REFORMAT.

38.3 Memory Mapping

The DOS occupies memory as shown by the following map:



38.4 Memory Tables

A number of entry point tables exist within the DOS. These tables consist of a group of jumps to the various routines made available to the user. These jumps allow the system to be changed without requiring the user to reassemble his programs.

The first entry point table is located between 01000 and 01377. It contains entry points to the routines in the loader (the loader itself, the basic disk read and write driver, and the interrupt handler) and the DOS file handling routines. It also contains in-line routines to increment and decrement the HL registers. These routines are coded in-line and constructed in a fashion to enable the A-register to contain the increment or decrement value and the entry point plus two entered for incrementation or decrementation by a number other than one.

The second entry point table is located between 010000 and 010066 and contains entry points to the cassette handling routines. The third entry point table is located between 013400 and 013452 and contains entry points to routines within the command interpreter. The availability of the command interpreter routines makes small command tasks easy to implement as can be seen by inspection of the assembly listings for the commands supplied with the DOS. See the Chapters on System Routines and Routine Entry Points for more details on the routine functions and entry point locations.

The major working table in the system is called the Logical File Table (LFT) and is located from 01544 through 01643. It contains all of the information required by the file handling routines for every file which is currently open (a maximum of three files may be open at any one time - logical files one, two, and three). Once the user has opened a file by its symbolic name, he deals with it by the logical file number under which it was opened. The LFT entry keeps a record of the segment currently being dealt with. If the record being accessed (specified by the LRN in the table) is within the current segment (determined from the BLRN and CSD in the table), the physical disk location can be determined from the information in the LFT (all LRN's in the LFT are system LRN's). Otherwise, the RIB must be read and a new current segment established. Note, however, that a maximum of only two disk accesses are necessary to randomly access any piece of information within a file and sequential accesses require only one disk access in most cases.

The LFT contains for each entry the following information in

the order shown (the number in parenthesis is the number of bytes used for the item):

```
PFN    (1) - Physical File Number
PDN    (1) - Physical Drive Number and Protection
LRN    (2) - Next LRN to be dealt with
BLRN   (2) - First LRN within the current segment
CSD    (2) - Current Segment Descriptor
RIBCYL (1) - Physical Disk Address of RIB (MSB)
RIBSEC (1) - Physical Disk Address of RIB (LSB)
MAXLRN (2) - Largest LRN referenced
LRNLIM (2) - Largest LRN allowed (obsolete)
BUFADR (1) - Current controller buffer address
XXXXXX (1) - Not used
```

There are actually four LFT entries to correspond to buffers 0-3 in the disk controller. However, the first entry (logical file zero) is reserved for system usage because the DOS needs a buffer into which it can read the RIB if it is necessary to determine a new current segment when a given access is made. This need is only critical on writes when the buffer contains the information to be written to the disk and the system must then read the RIB into a different buffer. On reads, the user's data will always be the last item to be read and logical file zero can be used. One must exercise caution in the use of logical file zero, however, since an access involving a different logical file may cause logical file zero's disk buffer to be loaded with a RIB. Also, the zeroth disk controller buffer is always used by the system loader in transferring data to memory. This last fact implies that the user may load an overlay or chain to another program without any of the standard (one through three) logical files being perturbed in any way. The other thing that is special about logical file zero is that CLOSEs have no effect when issued on it. This means that neither space deallocation nor updating of the protection field occur when logical file zero is closed. This is true whether the close is done by explicitly calling CLOSE$ or implicitly by calling other system routines (e.g. PREP$, LOAD$, RUN$, etc.)

The DOS loader uses a set of locations in memory locations 4 through 022 to perform the functions of an LFT entry during the loading process. It knows, however, that an object file is always sequential and does not have to have the accessing generalization of the main file handling routines. This is the main factor in the small size of the DOS loader. The file handling routines also use these low memory locations for temporary storage of a specified LFT entry to eliminate having to continually index into the LFT. Also, since the basic disk read and write routines use

location 5 to indicate which drive is to be used, having the LFT temporarily stored in the low memory locations automatically selects the correct drive for use.

38.5 The Command Interpreter

The Chapter on Operator Commands of this manual describes the operation of the DOS from the system console. When the command interpreter is entered, it checks to see if a program has been set to be automatically executed. If it has, the program is loaded and executed (unless the KEYBOARD key is depressed). Otherwise, the command interpreter attempts to obtain a command line from the keyboard. While it is waiting for this line, the command interpreter runs a test on the disk controller buffer memory while checking the keyboard ready status bit. When the keyboard becomes ready, the test is stopped and the keyin routine entered (which will get the character that made the keyboard become ready and then key in the rest of the command line - note that even striking just the CANCEL key will stop the disk buffer memory test).

The command interpreter resides in locations 012400 through 020000. Actually, the area between 017000 and 020000 is used as an overlay area by many of the commands supplied with the system, thus eliminating the need to reload the DOS after the execution of every command. The interpreter keys a command line into a place in the data area of the DOS called MCR$ (Monitor Communication Region, locations 01400 through 01543). It then scans this line with lexical scanning routines that are made available to the user through the command interpreter entry point table. The # (which causes entry into the debugging tool), is handled as a special case. Otherwise, the first name given is opened as logical file zero and that program loaded and executed. The program that is loaded has access to the command line in MCR$ and thus programs may be parameterized by information given after the program name.

The command interpreter scans up to four file specifications from the command line. The scan is terminated by a semicolon (;) or END-OF-STRING (015). The file specifications are entered in a normalized symbolic form into the corresponding logical file table entries. Note that this is not the normal logical file table information but that these locations are simply being used as a temporary storage for the symbolic information that has been lexically normalized by the command interpreter. The program loaded may simply check that all the necessary information has been supplied and/or supply any assumed portions, and then use this data as a name parameter to the file opening or creating routine. The opening routine has no difficulty using a name that is supplied as its parameter in the same locations as the logical

file table entry it is going to set up.

When a program receives control from the command interpreter after having been invoked via a command line from the keyboard, each of the LFT entries one through three (but not zero) contain the following:

```
DRCODE    (1) - Drive select code (described below)
0377      (1) - Indicates file is closed
FILENAME  (8) - File name specified (or eight spaces)
FILEEXT   (3) - File extension specified (or spaces)
DRSPEC    (3) - Logical drive specification (or spaces)
```

The drive select code contains one of the following:

```
0377      - No drive spec entered (DRSPEC is spaces)
0376      - Nonstandard, probably invalid, drive spec
Otherwise - The given logical drive number, in binary.
```

Note that most DOS system routines allow 0377 as a drive number, indicating "scan all drives". (This is the reason why 0377 is the no-drive-specified code; programs need no longer treat this as a special case). Example program usage: (prepare file specified in LF2 file spec, defaulting extension to TXT)

```
LFT      EQU   01544
LF2      EQU   32
TXT      DC    'TXT'          Default extension
NAMREQ   DC    011,0,013,11,023,'NAME REQUIRED',015
BADDRI   DC    011,0,013,11,023,'BAD DRIVE',015
START    MLA   *LFT+LF2+10    Get first byte of ext.
         CP    ' '            Check if it's blank
         LEL
         LDH
         HL    TXT            Page
         LC    3
         CTZ   BLKTFR         If Blank, default to "TXT"
         MLA   *LFT+LF2+2     Make sure name was given
         CP    ' '            And if not, get address
         HL    NAMREQ         of "NAME REQ" message
         JTZ   CMDAGN         And return with it
         MLA   *LFT+LF2       Check drive select code
         CP    0376           for validity and if bad
         HL    BADDRI         give "INVALID DRV" message
         JTZ   CMDAGN
         LCA                  Get drive # (N,0377) into C
         DE    LFT+LF2+2      DE => name, extension
         CALL  PREP$          And it's done.
```

Note that programs which receive control upon DOS startup (programs which have been set for AUTO-execute) do not generally receive control via the command line mechanism and hence the contents of both MCR$ and the LFT are initially undefined when such programs are entered. Any program which is to be set for automatic execution will therefore have to take this factor into consideration, and perhaps make special provisions as it may require. (A utility program called AUTOKEY exists which remedies this situation, and is released with the DOS).

CHAPTER 39. INTERRUPT HANDLING

39.1 Scheduling

When the system is loaded with the bootstrap function (RESTART depressed), the following set of CALL instructions (or their equivalent) are loaded into the area between 01000 and 01377 (just above the main entry point table for the DOS):

```
INTRPT  DI                      Disable interrupts
        BETA                    Use BETA mode
INT0    CALL  RETURN            Do the four
INT1    CALL  RETURN               one millisecond
INT2    CALL  RETURN               routines
INT3    CALL  RETURN
        MLA   *INTSCN           Rotate to the
        AD    6                    next one of the
        LMA                        four millisecond
        AD    INT4-6               routines
        LLA                     HL = CALL address
        PUSH                    Jump to the
RETURN  RET                        next CALL
.
INT4    CALL  RETURN            Four millisecond
        JMP   INTRET
INT5    CALL  RETURN
        JMP   INTRET
INT6    CALL  RETURN
        JMP   INTRET
INT7    CALL  RETURN
        XRA                     Reset the scan
        MSA   *INTSCN              pointer
INTRET  ALPHA                   Back to ALPHA mode
        EI                      Enable interrupts
        RET                     Back to the background
```

Foreground routines are executed by being called by one of the above CALL instructions, and run only in BETA mode with interrupts disabled. Note that the only way for a foreground routine to return to the scheduler is via a RETURN instruction. Therefore, the routine must always leave the subroutine call

address stack in the same state it was in when the routine was entered. This means that a foreground routine can not call a subroutine that then returns to the interrupt scheduler because this would leave the stack with an address it did not previously have. Subroutines that must wait for another interrupt must be handled by storing the return address into a memory location somewhere (usually most conveniently into the address portion of a jump instruction, as it turns out).

39.2 Process Initialization

When bootstrap occurs, a return is stored in location zero to cause interrupts to have no effect. Whenever an interrupt routine is activated, however, locations 0, 1, and 2 are loaded with a jump instruction to the label INTRPT in the preceding code. This activates the interrupt scheduler.

Once the jump instruction has been stored, the interrupt scheduler is executed every millisecond. Note that initially none of the CALLs would have any effect, since they all call a RETURN instruction. When an interrupt driven routine is initiated, however, its address is stored into the address portion of the CALL instruction, causing that routine to be executed. Interrupt driven routines are always initialized from the background program by the routine SETI$, which stores the address given in the D and E registers (MSB and LSB respectively) into the CALL instruction whose number is given in the C register (the number corresponding to the digit shown in the labels on the CALL instructions in the preceding code). The first four CALLs are executed every millisecond and the second four CALLs are rotated in execution causing any particular one to be executed only once every four milliseconds. Since the interrupt scheduler is entered every millisecond, the execution time for the one four millisecond and four one millisecond routines should not total more than one millisecond average to prevent an interrupt from being dropped.

The execution of the foreground process can be stopped in two ways. There is a background routine called CLRI$ which simply sets the D and E registers to the label RETURN in the preceding code and executes the SETI$ routine. There is also a foreground routine called TP$ which will be discussed after the explanation of process state changing.

39.3 Process State Changing

Once an address has been stored in a particular CALL instruction, the same location will be entered upon each execution of that CALL. This location is called the state of the foreground process. A routine exists above the interrupt scheduler (still below 01400) which allows the state of the process to be changed to the location following the call of that routine. The routine, shown below, is called CS$ for Change State:

```
CS$     POP                 DE = the address
        LEL                    after the CALL
        LDH                    instruction
        POP                 HL = the address
        PUSH                   of the CALL to
        CALL   DECHL           this process
        LMD                 Change the address
        CALL   DECHL           in the CALL to
        LME                    this process
        RET                 Back to the scheduler
```

This routine obtains the new state address by popping the stack. It assumes that after doing that, the stack is in the same state it was when the process call was executed, and thus can obtain the location of that call by popping the stack again. (This implies that CS$ should not normally be called from a subroutine within an interrupt-driven process, but only from "level zero" of the foreground process). It does this and stores the new address in the process call. It then executes a RETURN to give control back to the interrupt scheduler, thereby causing execution of that specific foreground process to wait until the next time the process call is executed.

A routine called TP$ exists which simply loads the D and E registers with the location RETURN in the interrupt scheduler code shown earlier and jumps to the second POP instruction in CS$. This routine is jumped to (not called) and, as mentioned before, terminates the execution of the foreground process.

At this point an example is appropriate. To simplify the discussion, it will be assumed that the process CALL has been initialized to the location LABEL1. The following routine decrements a memory location called COUNT until it becomes zero

and then changes its state to the location LABEL2, which waits for the keyboard status bit to be set and then obtains the character entered and continues with processing. Note that this has the effect of causing a delay of the number of milliseconds equal to the number that was initially in COUNT before continuing on to checking the keyboard and processing the character.

```
LABEL1  MLA    *COUNT          Get the count
        SU     1               Decrement it
        LMA                    Update memory
        RFZ                    Back to the scheduler
        CALL   CS$             Change state if zero
LABEL2  LA     0341            Then start checking
        EX     ADR                the keyboard
        IN                     Wait for KBD
        ND     2                  ready
        RTZ                    Back to the scheduler
        EX     DATA            Unless KBD ready
        IN                     Then get the key
                               Etc.
```

The following is a narrative of what takes place. It will be assumed that interrupt zero (INT0) has been initialized and the jump instruction to INTRPT stored in locations 0, 1, and 2. Upon occurrence of the next interrupt a jump to INTRPT occurs, interrupts are disabled, and the processor switched to BETA mode. A CALL to LABEL1 is then executed by the instruction at INT0. LABEL1 loads the A-register with the contents of COUNT, decrements it, and stores the result back into COUNT. If the result is not zero, the return is executed and the rest of the CALLs in the interrupt scheduler are executed, the processor switched back to ALPHA mode, interrupts enabled, and control passed back to the program that was interrupted. If the result is zero, CS$ is called. It gets the location LABEL2 by popping the stack into the DE registers. It then gets the location INT1 by popping the stack into the HL registers but leaves this value on the stack. It then stores the DE register values (equal to the address LABEL2) into the CALL at INT0 and returns, causing execution to continue at INT1. When the next interrupt occurs, the CALL at INT0 will be to LABEL2.

39.4 Timing Considerations

As mentioned before, the programmer must be careful with the amount of time he uses when constructing interrupt driven routines. Since the interrupt scheduler is entered every millisecond, the total execution time of the four one millisecond calls and the one four millisecond call must not average over one millisecond if no interrupts are to be missed. Because of this time restriction, the calls that are rotated in execution were constructed to allow processes which do not require the higher rate to not impose as much overhead on the system. When one is constructing a foreground process and discovers that its execution time is becoming excessive, he must break it down into several states with each state using a more appropriate amount of time. Note that the interrupt scheduler itself uses 130 microseconds when there are no processes active.

(Timings given here and elsewhere relate to the Datapoint 1100 and 2200 Version II processors. Users with Datapoint 5500 computers (and not concerned with downward compatibility with Datapoint 1100 and 2200 processors) will find that its increased speed allows less restriction on the execution time of foreground driven processes, assuming that the 5500 DOS is running in single partition mode.)

The 1100, 2200, and 5500 each contain a crystal controlled clock which causes an interrupt signal every millisecond plus or minus 50 nanoseconds (.005%). When this signal occurs, a flag within the processor, called Interrupts Pending, is set. Upon the occurrence of an instruction fetch cycle when interrupts are enabled and Interrupts Pending is set, the processor clears Interrupts Pending and executes a CALL to location zero instead of performing the normal instruction fetch. This implies that the processor buffers interrupts one deep since Interrupts Pending will remember the occurrence of an interrupt until they are enabled. Note that there is a delay between the actual time when the one millisecond signal occurs and the time when the CALL to location zero is performed. This delay is equal to the length of time between the occurrence of the interrupt signal and the occurrence of a fetch cycle when interrupts are enabled. Since the interrupt signal is asynchronous to when the background program will have interrupts disabled and even to when fetch cycles occur, jitter in the execution of the interrupt scheduler with relation to the actual occurrence of the interrupt signal is introduced. This jitter is of prime concern when dealing with interrupt processes and its sources and analysis for purposes of program construction are treated in the following paragraphs.

There are two major sources of interrupt execution jitter. The first is interrupts being disabled. The background program must disable interrupts whenever it has the system in a state that cannot be restored by the interrupt scheduler. The interrupt scheduler assumes that the background does not use the BETA mode of the processor and, therefore, that it can be used without being restored when control is returned. Because of this, the background program must have interrupts disabled whenever the BETA mode of the processor is being used.

The other system state that cannot be saved is that of the I/O devices. When interrupts are active and the interrupt driven routines are performing input or output operations, the background routines must either not do any I/O or must disable interrupts when dealing with a device. If a background routine addresses a given device without first disabling interrupts, it could be interrupted before getting around to using that device and the interrupt routine could address some other device. When control is passed back to the background routine, it will proceed with its I/O operation thinking that the device it addressed is still addressed, whereas the device that the interrupt routine accessed is the one actually addressed and confusion will occur. Therefore, any I/O operations performed by the background program must have interrupts disabled from before the time the device is addressed until after it is used.

Care must be exercised when disabling interrupts in the background program to prevent the loss of an interrupt. Since Interrupts Pending is only one bit of information, the occurrence of another interrupt signal before the previous one is processed will result in the state of Interrupts Pending not changing (since it will simply be set again and it is already set) and, therefore, the occurrence of the second interrupt will not be reflected in the state of the processor. This means that if interrupts are disabled for more than one millisecond, an interrupt will be dropped. In practice, interrupts should not be disabled in the background for more than a few hundred microseconds for the following reason:

> Suppose an interrupt process is active which is taking characters from a device at the rate of 700 per second. This implies that a character must be taken from this device on the average of one every 1.4 milliseconds (if the device contains a one character buffer). Suppose further that the interrupt process polled the device just before the next character became available. At this point, the process has about 1.4 milliseconds to

get the next character before it will be overstored by the following and cause a loss of data. Normally, if interrupts were enabled, the interrupt process would poll the device about 1.0 milliseconds later and get the character with time to spare. However, if the background routine disabled interrupts for 500 microseconds just before the next interrupt occurred, the interrupt process would not be executed soon enough and a data character would be lost.

As the above example shows, the actual execution time of the interrupt processes can be caused to jitter due to the background routine disabling interrupts. The worst case jitter is exactly equal to the maximum amount of time the background routines disable interrupts. The DOS routines disable interrupts no longer than 200 microseconds. The maximum time tolerable is equal to the difference between the time between interrupts (1000 microseconds) and the minimum time between necessary interrupt process executions (1400 microseconds in the case above).

Another source of jitter can be in the execution time of the foreground processes themselves. The jitter time for interrupt zero is exactly equal to that due to interrupts being disabled. However, interrupt one is not executed until after interrupt zero and if interrupt zero consumes a different amount of processor time on each interrupt, interrupt one's execution time will vary with respect to when interrupt zero started execution. This is an additional jitter factor which must be calculated for interrupt one. The same is true for the interrupts that follow, but they vary even more since the start of each succeeding interrupt process depends upon the total of the execution times of all of the proceeding interrupt processes.

39.5 DOS Usage

The DOS itself (i.e. excluding DOS functions and command programs running under the DOS) uses the interrupt facility in only two places. One is the debugging tool's dynamic P-counter display (if it is turned on; uses interrupt zero) and the other is for the cassette handling routines when used (uses interrupt one). Both of these routines introduce a maximum of 400 microseconds of jitter and consume an average of 150 microseconds of processor time (with peaks of 500 microseconds).

Users with Datapoint 5500 computers and running the full 5500 DOS must consider other details relating to timing of foreground routines, particularly on foreground routines that deal with non-DOS supported I/O devices. These details will be dealt with more fully in the DOS System Manuals for the appropriate DOS.

CHAPTER 40. SYSTEM ROUTINES

40.1 Parameterization

Parameters are passed to the subroutines through the registers. In the discussion of these parameters, the following abbreviations will be used:

LFN - Logical File Number times 16 (16, 32, or 48)
LRN - Logical Record Number (the user's LRN)
PFN - Physical File Number
LFT - Logical File Table

also:

Drive Number - indicate a logical drive number (0 through N). (N varies with the DOS in use, but in general will be 2**X-1; typically 3, 7, or 15). In some routines, 0377 is used to indicate that all drives are to be checked.

Name- the address of a field containing exactly eleven bytes. The first eight bytes are the file name and the last three bytes are the file extension by command interpreter convention. The name characters may be any eight bit combinations except the first character must not be a 0377. The command interpreter requires that the first and ninth characters be letters and that the remaining be letters or digits with trailing spaces.

40.2 Exit Conditions

When a routine is used, it can either perform the expected action or not. In the second case, some indication must be made that the expected action did not occur. This is achieved by the condition flags in the processor being set in a special manner or by control being transferred to a trap location instead of being returned via the subroutine mechanism. The 'Exit conditions' section of each subroutine description shows the register contents and condition flags of interest when the routine returns.

40.3 Error Handling

There are fatal and non-fatal errors. Fatal errors suggest that the program is hopelessly confused and the only recourse is to display what the problem appears to be and reload the operating system. This is usually the result of a call being incorrectly parameterized or the system tables on the disk being unusable. The messages displayed are explained in the Chapter on Error Messages.

Non-fatal errors concern various conditions such as parity failures in the user's data, records of illegal format, violations of a file's protection, or physical end of cassette. In some cases these conditions can be detected upon the routine's exit as explained in the section on Exit Conditions. In the other cases, control is passed to a specified location (to the error message routine if no location has been specified by the user) instead of being returned via the normal subroutine exit mechanism.

There are actually two sets of traps. The first deals with the disk routines and the second deals with the cassette routines. The disk routine traps are described under the section on File Handling Routines and the cassette routine traps are described under the section on Cassette Handling Routines. The 'Traps' section of each subroutine description indicates what conditions will cause the relevant traps. These traps are referenced by mnemonics which are defined in the section where the trap setting routines are described (the section on TRAPS and TTRAP$).

40.4 Foreground Routines

Section 4 contains a complete discussion on the functioning and use of the foreground handling and should be consulted for an understanding of the following routines.

40.4.1 CS$ - Change Process State

CS$ changes a foreground routine's state. It is called by the executing foreground routine and causes its execution address to be changed to the address following the CALL CS$. Execution will not continue at the new address until the next interrupt occurs. CS$ is normally called from the outermost level (level 0) of an active foreground process.

Entry point: 01033

Parameters: on subroutine stack - see the Chapter on Interupt

Handling

Exit conditions: return is made to the scheduler

40.4.2 TP$ - Terminate Process

TP$ deactivates the process called by storing the address of a return instruction in the process call. TP$ is jumped to, not called. TP$ is invoked from the outermost level (level 0) of an active foreground process.

Entry point: 01036

Parameters: on the stack - see the Chapter on Interrupt Handling

Exit conditions: no exit, return to Interrupt Scheduler

40.4.3 SETI$ - Initiate Foreground Process

SETI$ activates the interrupt process specified by the number in the C register (0-7) by storing the address given in the D and E register into the CALL instruction for that process. Interrupt processes zero through three are executed every millisecond while four through seven are executed every fourth millisecond.

Entry point: 01041

Parameters: C = process number (0-7)
DE = address of foreground process

Exit conditions: B,D,E unchanged
H,L = 0

40.4.4 CLRI$ - Terminate Foreground Process

CLRI$ deactivates a foreground process by storing the address of a return instruction into the process call specified by the number in the C register (0-7).

Entry point: 01044

Parameters: C = process number (0-7)

Exit conditions: B,D,E unchanged
H,L = 0

40.5 Loader Routines

There are two levels of disk handling routines. This section describes the lower level routines which reside in the loader and require numbers physically describing the drive, cylinder, sector, buffer, and file. The section on File Handling Routines describes the upper level routines.

INCHL and DECHL are described in this section only because they are used by the DOS at all levels and because these two routines are loaded as part of the bootblock. In general, the other routines described in this section are not used by typical user programs; most user programs will be better served by the higher level routines described in the section on File Handling Routines.

40.5.1 BOOT$ - Reload the Operating System

BOOT$ loads and executes the operating system (PFN 0 on logical drive 0). This action does not affect the interrupt handling facility between 01000 and 01377. Since BOOT$ requires that the operating system always be loaded from specifically drive zero, BOOT$ should normally only be used in cases where EXIT$ is unusable, for example if the disk handling routines have been overstored. BOOT$ does not close any files before reloading the DOS.

Entry point: 01000

Parameters: none

Exit conditions: does not return

40.5.2 RUNX$ - Load and Run a File by Number

RUNX$ loads the physical file specified and begins its execution. If the file cannot be loaded, a jump to BOOT$ occurs.

Entry point: 01003

Parameters: A = PFN
C = Drive Number

Exit conditions: does not return

40.5.3 LOADX$ - Load a File by Number

LOADX$ loads the physical file specified and returns with the starting address in HL if the load was successful.

Entry point: 01006

Parameters:
A = PFN
C = Drive Number

Exit conditions: Carry false: HL = Starting address of file
Carry true: A=0 if file does not exist
1 if drive off line
2 if directory parity fault
3 if RIB parity fault
4 if file parity fault
5 if off end of physical file
6 if record of illegal format

40.5.4 INCHL - Increment the H and L Registers

INCHL increments the sixteen bit value in the HL registers by one. If the routine is entered at INCHL+2, the sixteen bit value in the HL registers will be incremented by the number in the A register.

Entry point: 01011 (01013 for increment by A)

Parameters:
HL = number to be incremented
A = increment value if INCHL+2 used

Exit conditions: HL incremented
A equal to the H-register
B,C,D,E unchanged
CARRY condition undefined

40.5.5 DECHL - Decrement the H and L Registers

DECHL decrements the sixteen bit value in the HL registers by one. If the routine is entered at DECHL+2, the sixteen bit value in the HL registers will be decremented by negative the number in the A register (e.g., for decremention of 2, A is set to -2).

Entry point: 01022 (01024 for decrement by -A)

Parameters: HL = number to be decremented
A = decrement value if DECHL+2 used

Exit conditions: HL decremented
A equal to the H-register
B,C,D,E unchanged

40.5.6 GETNCH - Get the Next Disk Buffer Byte

GETNCH gets the character from the physical disk buffer location pointed to by low memory location DOSPTR (location 026) from the disk buffer currently selected and then increments the contents of the location DOSPTR.

Entry point: 01047

Parameters: DOSPTR = disk buffer address (0-255)

Exit conditions: A = character from disk buffer
(DOSPTR) = (DOSPTR)+1
B,C,D,E,H,L all unchanged

40.5.7 DR$ - Read a Sector into the Disk Buffer

DR$ causes a sector to be transferred from the disk to one of the disk controller buffers. The drive number is given in the least significant bits (the others are ignored) of location TFT+PDN (5). (The number of bits ignored depends upon the particular DOS in use). The physical disk address (LSB) is given in the E register and the physical disk address (MSB) is given in the D register. The disk controller buffer number times sixteen is given in the B register. Interrupts are disabled by this routine a maximum of 100 microseconds.

Compatibility note: Here the user should be reminded that the physical disk address format will vary; the user's program should not make assumptions regarding this format if the program is to be transportable between different DOS. The most significant byte is generally a cylinder number, and the least significant byte is a sector address within a cylinder. This least significant byte will generally be the more at variance among DOS. In general, the only safe way to insure a valid, proper physical disk address (PDA) is to get it as a returned item from a system routine (POSIT$ or one of the DOS FUNCTIONs, to be described later). User program generation of or manipulation of physical disk addresses is strongly discouraged.

DR$ tries between four and ten times to read a record (depending upon the disk drive type in use), if parity faults are detected, before giving an abnormal exit status. Note that since this routine is used by all of the higher level routines, all disk reads performed by the disk operating system try to read a record that shows parity problems that same number of times before giving up.

Entry point: 01052

Parameter: B = 16 times buffer number (0,16,32,48)
D = physical disk address (MSB)
E = physical disk address (LSB)
TFT+PDN (at loc 5) = logical drive number

Exit conditions: B,D,E,TFT & PDN all unchanged
Carry false if read successful
Carry true and Zero false if drive off line
Carry true and Zero true if parity fault

40.5.8 DW$ - Write a Sector from the Disk Buffer

DW$ causes the contents of one of the disk controller buffers to be transferred to a sector on the disk. If the write protection on the specified drive is enabled, DW$ will beep continuously until the protection is disabled.

There are two types of write protection in the disk operating system. The first type is a physical protection that is part of the disk drive hardware which will cause DW$ to beep if set. The second type of write protection is a logical protection that is connected with each file on a disk. A bit exists in the directory entry for each file which, if set, will prevent the higher level routines (for example, WRITE$) from calling the DW$ routine. It is important not to confuse these two types of write protection. All references to write protection that follow refer to the logical protection on each file and not to the physical protection on the drive itself.

In DOS, DW$ uses the write/verify mode of the disk controller. This implies that all writes made by these disk operating systems use this mode of writing. As in the DR$ routine, several tries will be made if parity faults occur before abnormal exit will occur. In all other respects, DW$ is similar to DR$.

Entry point: 01055

Parameters: B = 16 times buffer number (0,16,32,48)
D = physical disk address (MSB)
E = physical disk address (LSB)
TFT+PDN (at loc 5) = drive number

Exit conditions: B,D,E,TFT & PDN unchanged
Carry false if read successful
Carry true and Zero false if drive off line
Carry true and Zero true if parity fault

40.5.9 DSKWAT - Wait for Disk Ready

DSKWAT waits for disk ready, controller ready, no disk I/O transfer in progress, and drive online to all be true. If the drive is not online, return is made with the carry flag true, the zero flag false, and interrupts enabled. Otherwise, exit is made with interrupts disabled. This routine is obsolete and is not available under some systems (e.g. PS). Therefore, it should no longer be used.

Entry point: 01060

Parameters: none (drive checked is the selected drive)

Exit conditions: explained above
B,C,D,E,H,L unchanged

40.6 File Handling Routines

A file is dealt with as a logically contiguous and randomly accessible space. The file being used is specified by its symbolic name . The LRN being dealt with within that file is determined by a two-byte number kept within the system (LRN in the LFT). When a file is opened, this number is set to two.

A bit of explanation may be called for here. This two corresponds to user logical record number zero; the LRN in the LFT is the system LRN. System LRN zero is the primary RIB for the file and system LRN one is the RIB backup. System LRN two is the first user data sector. It is important to recognize this distinction between system and user logical record numbers. All logical record numbers supplied to system routines (e.g. POSIT$) are <u>user</u> logical record numbers. These are converted to <u>system</u> logical record numbers before being used by the DOS or placed into the LFT.

After each record access (READ$ or WRITE$), LRN is

incremented. Thus, for sequential accesses, the user does not actually specify which record he is dealing with. However, a routine exists which allows the LRN to be changed to any value between zero and the upper limit on the file, POSIT$, providing a random access facility. (This upper limit depends upon the DOS in use). Note that, since no end of file mark is intrinsic to the system, the user must provide his own special data record to denote an end of file during sequential accesses.

If a user wishes the option of processing his files using the standard DOS utility programs (SAPP, LIST, REFORMAT, etc.) then his EOFmark should follow DOS EDITOR conventions:

1) The first six user data bytes in the EOFmark sector are binary zeros.
2) The seventh user data byte in the EOFmark sector is a binary three.

For example: Assume the user has moved the last data record to be written to the appropriate disk buffer. (The terminating 03 is assumed to be there also). The following sequence will write the final record and create a valid DOS EOFmark:

```
        LB      LFn         Specify output LFN
        CALL    WRITE$      Write last data record
EOFPUT  XRA                 Set A to binary zero
        CALL    PUT$        Output zero to buffer
        CP      9           6 Bytes written?
        JTC     EOFPUT      Repeat as needed
        LA      3           Set A to binary three
        CALL    PUT$        Last byte of EOFmark
        CALL    WRITE$      Write the EOFmark
        CALL    CLOSE$      And close the file
```

40.6.1 PREP$ - Open or Create a File

PREP$ searches the directory or directories specified for the given name. If the name is found, the file is simply opened for use as the specified logical file number. Otherwise, a new file having the name specified will be created. If a new file is created, an end of file by GEDIT convention (six zeros followed by an 003) is written in logical record zero. Whether the file is simply opened or is created, the information describing it is stored in the LFT entry specified so that all subsequent references to that file by its LFN will be able to deal with the correct locations on the disk. If the LFT entry specified is

already in use when PREP$ is called, the file that the entry specifies will be closed (see the section on CLOSE$) and then the new file opened in its place.

DE is normally the address of an 11-byte string which is the name of the file being specified (as explained before under the section on PARAMETERIZATION. However, if the D register is zero then the E register contains the physical file number. The ability to reference files by number makes it possible to avoid the substantial time required to search the directory for a name. If the PFN is already in use, a 'SPACE' trap will occur. Otherwise, the file of that number will be created. When a file is created by number, its name in the directory consists of all 0377 characters, preventing it from being accessed symbolically or being listed by the catalog listing command. When a PFN is supplied, a particular drive must be specified (0377 may not be specified as a drive number).

Entry point: 01063

Parameters:
B = LFN
C = Drive Number or 0377
DE= Name or D=0 and E=PFN
If PFN given, C must not be 0377.

Exit conditions: B = LFN; other registers indeterminate

Traps:

SPACE	if a new file must be allocated and no space is left or no more directory entries are available
OFF-LINE	If the DRIVE specified is off-line.

40.6.2 OPEN$ - Open an Existing File

OPEN$ is similar to PREP$ except for the action taken if the file specified does not exist. In this case, return is made with the Carry condition true (return is made with it false if the file exists). Action is similar if a PFN is supplied instead of the name. If the PFN specified exists, the file is opened and return is made with the Carry condition false. Otherwise, return is made with the Carry condition true.

Entry point: 01066

Parameters: same as for PREP$

Exit conditions: B = LFN; other registers indeterminate
Carry true if the file is non-existent

Traps: none

40.6.3 LOAD$ - Load a File

LOAD$ opens the specified file as logical file zero and then calls the system loader to load it into memory. Exit is made with the Carry condition set if the file is non-existent, or if the drive specified (if any) is off line. If the load is successful, return is made with the starting address in the H and L registers.

Entry point: 01071

Parameters: same as for PREP$ (except B not required)

Exit conditions: B = LFN (always zero)
HL = starting address if good load
Carry true if file non-existent or drive off-line

Traps:

OFFLIN	drive went off line after loading began
RPARIT	file contains parity fault
RANGE	loader ran off end of file
FORMAT	record of bad loader format

40.6.4 RUN$ - Load and Run a File

RUN$ opens the specified file as logical file zero and then calls the system loader to load it into memory. Return is made to following the call if the name specified cannot be found in the directory or directories specified. If any loading errors occur, the operating system is reloaded. Otherwise, control is transferred to the starting address given by the loader.

Entry point: 01074

Parameters: same as for PREP$
(except that B is not required)

Exit conditions: returns if name not in directory
operating system reloaded if bad load
otherwise, control is passed to the
starting address of the new file.

Traps: none

40.6.5 CLOSE$ - Close a File

When new space is allocated for a file, a large contiguous piece (up to one full segment) is taken in an effort to keep the file as physically contiguous as possible. When this allocation takes place, a flag in the LFT, called the new space allocated flag, is set. The LFT also contains a number which is the largest LRN referenced while the file was open. When CLOSE$ is called, the file is physically truncated after the largest LRN referenced, if the new space allocated flag is set. Thus, if only a few records of the new space allocated have been used, the rest of the space is freed for use in other files. However, if all of the space is used, the file will consist of a large amount of physically contiguous space. Note that if CHOP$ was called with the D register set to -1 (0377), and the LRN in the LFT has not been changed, a call to CLOSE$ will delete the entire file and remove its entry from the directory.

After the file has been truncated, if necessary, CLOSE$ then writes the copies of the protection bits and old file length limit field that are in LFT entry back into the directory. Therefore, one only needs to change these entries in the LFT and then close the file to have them changed in the directory. This is the basis for the functioning of the CHOP$ and PROTE$ routines. Since the protection bits and old file length limit field are not changed on the disk until the CLOSE$ routine is called, if one changes these numbers and then, for some reason, reloads the system without calling the CLOSE$ routine (by depressing RESTART before the file is closed, for example) the disk will retain the old values.

After the protection and file length limit have been stored in the directory, CLOSE$ then vacates the LFT entry specified. This is achieved by storing an 0377 in the second byte of the entry (this is the drive number and 0377 denotes that the LFT entry is not in use). CLOSE$ simply returns if the LFT entry is not in use.

Entry point: 01077

Parameters: B = LFN (16,32,48; 0 => NOP)

Exit conditions: B = LFN; other registers indeterminate

Traps: none

40.6.6 CHOP$ - Delete Space in a File

CHOP$ sets the maximum LRN value kept in the LFT and sets the new space allocated flag if no protection is set. If the CLOSE$ routine is called after the call to CHOP$ without the LRN being changed, the space after the specified LRN will be physically deleted from the file, making it free again for allocation by the system. Note that if the D register is set to -1 (0377) upon entry to CHOP$, calling the CLOSE$ routine will completely delete the file from the system (removing its entry from the directory as well as freeing all of its space). When an entry is deleted from the directory, all sixteen bytes of the directory for that entry are set to 0377. This is the same value for an unused directory entry that is set by the system generation program.

CHOP$ changes the MAXLRN field in the LFT to the LRN supplied as the parameter. (Note: CHOP$ makes provision here for the two RIBs and biases the LRN supplied by the user by two before placing it into the LFT MAXLRN field). Remember that calling CHOP$ only affects the LFT entry and that no physical change on the file is effected until CLOSE$ is called.

Entry point: 01102

Parameters:
B = LFN
DE = LRN if D not less than zero
d = -1 (0377) to delete entire file

Exit conditions: B = LFN; other registers indeterminate

Traps:

RANGE	DE not less than MAXLRN referenced
DVIOLA	delete protection is set
WVIOLA	write protection is set

40.6.7 PROTE$ - Change the Protection on a File

PROTE$ changes the file protection bit and/or upper file length limit copies that are kept in the LFT. The protection bits, given in the C register, are changed only if the least significant bit of the C register is a one. The old upper file length limit field is changed only if the sign bit of D is one on entry. Therefore, setting the number to zero prevents the limit field from being changed. Note that the file length field is obsolete and is no longer used by the DOS; it is maintained for future use, probably as a file type designation field.

Entry point: 01105

Parameters: B = LFN
C = new protection:
C0 = 1 for protection change
C6 = 1 for write protection
C7 = 1 for delete protection
DE = new LRN limit field; 0 for no change

Exit conditions: B = LFN; other registers indeterminate

Traps: none

40.6.8 POSIT$ - Position to a Record within a File

POSIT$ positions the file logically to the user LRN given. If the user LRN given is -1, the current value in the LFT is used for positioning the head and the LFT entry is not changed. Note that positioning to user LRN zero performs a logical 'rewind' of sequential files.

Entry point: 01110

Parameters: B = LFN
DE = LRN (use LRN from LFT if DE = -1)

Exit conditions: B = LFN
D = Physical Disk Address (MSB)
E = Physical Disk Address (LSB)
ZERO FALSE: DE are valid, position was valid
ZERO TRUE: DE are invalid, specified sector not in allocated space
other registers indeterminate

Traps: none

40.6.9 READ$ - Read a Record into the Buffer

READ$ causes the record, pointed to by the LRN in the LFT entry specified by the LFN given, to be transferred from the disk to the disk controller buffer that corresponds to the LFN given. The LRN is incremented by one after the read if it was successful. READ$ tries four times to read a record, if a parity fault is detected, before giving the trap. Attempting to read a record that is not physically allocated will cause the 'RANGE' trap.

Entry point: 01113

Parameters: B = LFN

Exit conditions: B = LFN; other registers indeterminate

Traps:		
	RANGE	LRN out of range
	RPARIT	record unreadable
	FORMAT	PFN or LRN in record incorrect
	OFFLIN	drive off line

40.6.10 WRITE$ - Write a Record from the Buffer

WRITE$ first takes the PFN and LRN values from the LFT entry specified by the LFN given and stores them into the first three bytes of the disk controller buffer that corresponds to the LFN given. It then transfers that buffer to the sector on the disk specified by the LRN in the LFT entry specified by the LFN given. The LRN is incremented after the write if it is successful. Note that all system routines use DW$ in writing records and hence try up to four times to obtain a good write, if a parity fault is detected, before giving the trap.

If WRITE$ tries to write a record which would not go in a place that has been physically allocated, it will automatically try to allocate more space. If the space is available, it is allocated and the write occurs. If there is no more physical space on the disk or if there are no more entries in the RIB available for the new segment descriptor, a 'SPACE' trap is given.

Entry point: 01116

Parameters: B = LFN

Exit conditions: B = LFN; other registers indeterminate
LRN = LRN + 1

Traps:		
	WVIOLA	file is write protected
	WPARIT	write/verify failure
	OFFLIN	drive off line
	RANGE	LRN < 0
	SPACE	explained above

40.6.11 GET$ - Get the Next Buffer

The LFT contains an entry called BUFADR (not to be confused with loc. 026 used by GETNCH) which points to a character in the disk controller buffer that corresponds to the given LFN. Each buffer contains 256 characters but since the system uses the first three bytes in each sector to store the PFN and the LRN of each record, the user has only 253 bytes available.

Whenever READ$, WRITE$, or POSIT$ are executed, they set the buffer pointer mentioned above to point to the third byte in the disk controller buffer associated with the given LFN (by setting the BUFADR field of the LFT entry to a three). Whenever GET$ is called, the byte pointed to by this pointer is fetched from the disk controller buffer and the pointer is incremented. If the byte being returned is not a valid user data byte (i.e. BUFADR was 0,1,or 2 on entry) then carry is true on return, and register A contains the specified byte of the buffer (which will be PFN or one of the LRN bytes.) Note that the next buffer is not read automatically from the disk; the pointer simply ends-around. Upon the first call of GET$ which returns carry true, the PFN will be obtained since it is contained in buffer location zero. The first three bytes may also be accessed by simply setting the buffer pointer contained in the LFT entry to the desired location.

Entry point: 01121

Parameters: B = LFN

Exit conditions: A = the byte obtained from the buffer
All other registers preserved
Carry true if location 0,1,or 2 accessed

Traps: none

40.6.12 GETR$ - Get an Indexed Buffer Character

GETR$ is similar to GET$ except that it uses the logical buffer address supplied in the C register instead of the physical buffer address in the LFT for the address of the disk buffer byte to return. Calling GETR$ has no effect on the buffer pointer kept in the LFT. The physical buffer location is obtained by adding three to the value given in the C register to skip past the system data in the first three bytes in the disk buffer. Thus the user is presented with a logical space within a record that is addressed from 0 through 252. Normally, GETR$ exits with the value in the C register incremented by one and the carry condition false. However, if the C register is between 253 and 255 (inclusive) upon entry, it will not be incremented and exit will be made with the carry condition true. In either case, the buffer byte located by the C register value plus three is returned in the A register. Therefore, the user may obtain any buffer byte with GETR$ but must remember to supply an address which is the physical buffer address minus three and remember not to assume that the C register will be incremented if he plans to access one of the first three physical bytes.

Entry point: 01124

Parameters: B = LFN
C = buffer location

Exit conditions: A = byte obtained
C = C + 1 if carry false
Carry true if 252 < C < 256
All other registers preserved

Traps: none

40.6.13 PUT$ - Store into the Next Buffer Position

PUT$ is similar to GET$ except that the byte presented in the A register on entry is stored into the buffer. Also, on return register A contains the physical address of the next byte to be accessed in the disk buffer. Carry is true if the byte stored was stored into the last physical location in the buffer. Here a reminder is appropriate: remember that in standard, EDIT-format records, the last two bytes (at least) of the buffer are not used, and an 03 occurring earlier in the sector indicates logical-end-of-sector. (A complete description of the format for EDIT-compatible text files can be found in the chapter describing the REFORMAT command.)

Entry point: 01127

Parameters: A = the byte to be stored in the buffer
B = LFN

Exit conditions: A as described above (physical address of next byte)
All other registers preserved
Carry true if location 255 was stored into

Traps: none

40.6.14 PUTR$ - Store into an Indexed Buffer Position

PUTR$ is identical to GETR$ except that the byte presented in the A register is stored into the buffer.

Entry point: 01132

Parameters: A = byte to be written
B = LFN
C = logical buffer location

Exit conditions: C = C + 1 if carry false
Carry true if 252 < C < 256
All other registers preserved

Traps: none

40.6.15 BSP$ - Backspace One Physical Sector

BSP$ decrements the LRN in the LFT entry specified by the LFN given and then executes POSIT$. No check is made to prevent calling BSP$ from backing P into a RIB. However, if one calls BSP$ and attempts to backspace back beyond system LRN 0 (user LRN -2, which is the master RIB) ZERO TRUE will be returned (as for POSIT$).

Entry point: 01135

Parameters: B = LFN

Exit conditions: B = LFN; other registers indeterminate

Traps: none

40.6.16 BLKTFR - Transfer a Block of Memory

BLKTFR moves the number of bytes specified in the C register (0 causes transfer of 256 bytes) from the memory location starting where HL points to the memory location starting where DE points. Note that since exit is made with HL and DE pointing after the last byte moved and C equal to zero, transfers of more than 256 bytes may be made by first setting C to zero, calling BLKTFR enough times to make the residual number of bytes to transfer less than 256, setting C to the residual number of bytes to be transferred, and then calling BLKTFR one last time. For example:

```
HL      SOURCE
DE      DEST
LC      0
CALL    BLKTFR
CALL    BLKTFR
LC      25
CALL    BLKTFR
```

will cause 537 bytes to be transferred from SOURCE to DEST.

Entry point: 01143

Parameters: C = number of bytes to be moved

(0 moves 256 bytes)
HL = source address
DE = destination address

Exit conditions: HL = HL + C (HL + 256 if C = 0)
DE = DE + C (DE + 256 if C = 0)
B = unchanged
C = zero

Traps: none

40.6.17 TRAP$ - Set an Error Condition Trap

There are eight non-fatal error conditions, concerning the disk operating system file handling facilities, that may be trapped by the user. If the trap corresponding to a certain error is not set by this routine, the system displays a pertinent message and reloads the system. Otherwise, control is transferred to the address specified when the trap was set, with the subroutine return address stack in the state it had before the calling of the file handling routine that caused the error condition.

The only disk errors that cannot be trapped are ones associated with the system tables on the disk. The occurrence of these errors causes the message

FAILURE IN SYSTEM DATA

to be displayed. The other errors that cannot be trapped have to do with: the LFT entry not being open when a routine which tried to use data from the entry was called, invalid logical file numbers, invalid drive numbers, invalid trap numbers, and invalid physical file numbers.

If a trap occurs during a call to READ$ or WRITE$, the logical record number (LRN) in the logical file table (LFT) is NOT incremented; if the user wishes to continue processing records past the one which caused the trap, he must increment the LRN in the LFT himself first.

TRAP$ sets the trap whose number is given in the C register to the address supplied in the D register (MSB) and E register (LSB). The trap is cleared by calling TRAP$ with D and E equal to zero. The trap is also cleared when the error condition occurs, at which time the B register will be loaded with the Logical File Number involved and control transferred to the indicated address.

In the following table, the mnemonic given after the trap number is the one used in the previous routine explanations. The capitalized lines are the messages displayed if the trap is not set.

0 - RPARIT - PARITY FAILURE DURING READ
A parity fault while reading a data record causes this trap.

1 - WPARIT - PARITY FAILURE DURING WRITE
A parity fault while writing a data record causes this trap.

2 - FORMAT - RECORD FORMAT ERROR
The physical file number or logical record number in the record read not matching the ones contained in the logical file table entry causes this trap. The physical position of a record is obtained from information in the retrieval information block and the PFN and LRN in the record are only checked to ensure that the drive is functioning correctly and that the user is not trying to read a record he has not written. This trap has nothing to do with the 253 data bytes provided to the user.

3 - RANGE - RECORD NUMBER OUT OF RANGE
During a read, an access below zero or to a record above the currently allocated space causes this trap. During a write, an access below zero causes this trap.

4 - WVIOLA - WRITE PROTECT VIOLATION
An attempt to write on, delete, or shorten a file with the write protection bit set causes this trap.

5 - DVIOLA - DELETE PROTECT VIOLATION
An attempt to delete or shorten a file with the delete protection bit set causes this trap.

6 - SPACE - FILE SPACE FULL
An attempt to allocate more space when either the disk is full or no more segment descriptor slots in the RIB are available causes this trap.

7 - OFFLIN - DRIVE OFF LINE
An attempt to use a drive that is either physically absent or not online causes this trap.

Note that the causes given for the various traps are the causes for DOS to issue the appropriate messages. Some of the DOS

Command programs also cause the issuance of some of these messages for related reasons. For example, several DOS Utilities indicate a RECORD FORMAT ERROR if the sector formatting of a file being processed does not follow GEDIT (or DOS EDITOR) standards. In cases such as this the above details are sometimes not valid descriptions of the problem; here the 253 data bytes encountered may have everything to do with the cause of the record format error.

Note also that FORMAT and RANGE traps are frequently the result of sequentially reading or otherwise processing a file which has no valid EOFmark, resulting in the program running off the logical end of the file.

Entry point: 01146

Parameters: DE = trap address
C = trap number

Exit conditions: register contents indeterminate

Traps: none

40.6.18 EXIT$ - Reload the Operating System

EXIT$ closes any logical files (one through three) that are open and then reloads the operating system. EXIT$ is the normal exit for all DOS programs. If drive zero is off line when EXIT$ is reached (or if the DOS is unloadable from there for any reason) an automatic drive switch occurs (indicated by a beep) and an attempt is made to load the DOS from the next drive in sequence. The automatic drive switch and beep is repeated until the DOS is successfully loaded. One jumps to this entry point.

Entry point: 01151

Parameters: none

Exit conditions: no exit

Traps: none

40.6.19 ERROR$ -- Reload the Operating System

ERROR$ is identical to EXIT$ in all respects except for the fact that jumping to ERROR$ will abort an active CHAIN (refer to the CHAIN command in this manual for more details). A user program would exit through ERROR$ if an error of severity suggesting aborting a CHAIN occurred.

Entry point: 01140

Parameters: none

Exit conditions: no exit

Traps: none

40.6.20 WAIT$ -- DOS Wait-a-While "NOP" Routine

This routine, after being called, returns with all registers, condition codes, and the stack preserved; in effect a "NOP". Normally, the return is immediate. If the Partition Supervisor is active, it time-slices to the other partitions before returning, a delay of several (maybe 10 or 20) milliseconds. This routine should be used in loops which wait for time non-critical conditions to occur (e.g. waiting for the keyboard operator to release the DISPLAY key), allowing the other partitions more processing time. I/O status, including in particular the device addressed, is subject to change on return.

Entry point: 01170

Parameters: none

Exit Conditions: Registers and condition codes unchanged

Traps: None

40.7 Keyboard and Display Routines

40.7.1 DEBUG$ - Enter the Debugging Tool

The debugging tool enables the programmer to load files by number, examine and modify memory locations, set break points, and execute sections of his program. This facility greatly simplifies the task of debugging machine language programs.

The debugging tool can be entered from the command interpreter by entering a single pound sign (#) on the command line or from the user's program by jumping to the entry point. When it is executing, two numbers are displayed vertically in the last column of the screen. The top number, consisting of five digits, is an address and the bottom number, consisting of three digits, is the content of that address. After these numbers are displayed, input is requested from the keyboard as indicated by a flashing cursor. Commands to the debugger are given in the form <n>X where <n> is any number of octal digits and X is a command character. The command is executed immediately upon depression of the command character key without waiting for the ENTER key (the ENTER character is a command in itself).

All keys that are not recognized are ignored with a beep signaling the rejection. The BACKSPACE key is ignored but since commands use only the lower eight or sixteen bits of <n>, errors in the entry of numbers can be corrected by striking several zeros and then entering the correct digits. Alternatively, the CANCEL key causes the current input line to be erased without changing the current address. Although display stops if the cursor runs off the screen during input, characters are still accepted.

The debugger maintains a current address that is usually displayed as the five digit number at the right of the screen. There are times, however, when the five digits at the right of the screen do not reflect the current address and caution must be exercised to avoid confusion as to the value of the current address. The ENTER key is normally used to change the current address, but depressing it without preceding it with any digits will cause the current address to be displayed. Therefore, if there is any doubt about the number being displayed on the screen, simply depressing the ENTER key will ensure that the current address is being displayed.

Whenever the debugger is entered either from the jump to the entry point or from a return from a break point or call command, a beep is given and the state of all of the alpha mode registers and condition flags is saved. The value initially displayed is the top of the stack at entry, unless DEBUG was entered from a DOS

DEBUG breakpoint; in this case the address displayed is the address where the breakpoint was set. In all cases, the stack is preserved as at entry and the current address is set to the address displayed at entry. This enables the user to tell exactly the state of his program when the debugger was entered. Whenever a memory location is called or jumped to, the state of all of the alpha mode registers and condition flags is restored from the values saved at entry. Since these values are saved in memory, the programmer can simply modify these locations to change the values used to initialize the state of the alpha machine before control is transferred.

The major debugging technique is the setting of break points at critical places in the program and the execution of portions of the program while checking the values of the registers and critical memory locations at each break. The debugger sets a break point by storing a jump instruction, to a special entry point in itself, in the current address and the following two locations. (Notice that setting break points less than three bytes apart is therefore not a good idea.) Before the jump is stored, the content of the memory locations to be used is saved in a table in the debugger. When the break point is reached, the memory locations are restored with their original contents. A maximum of four break points may be active at any one time. A command is provided for insuring that all break points have been restored. When a break point is executed, the current address is set to the first byte of the break point jump instruction. Since the J command causes a jump to the current address if no digits precede it, one can continue execution of the routine that was broken by simply depressing the J key. Execution will continue with the first byte that was overstored by the break point jump with the state of the alpha machine exactly like it was before the break occurred. Thus, the programmer can set a break point, start execution, examine the registers when the break occurs (since register viewing does not change the current address) and then depress the J key to continue execution. This technique allows him to practically single step his program.

ENTRY POINT: 01154

COMMANDS:

B - Set a break point at the location given or, if no number is given, at the current address. Caution should be exercised to insure that the current address is pointing to the desired location if it is used.

C - Execute a call to the number given or, if no number is given,

to the current address. The alpha machine state is loaded from the values saved in the debugger before the call is executed. A return to the call causes the debugger to be re-entered and the alpha machine state to be saved.

D - Decrement the current address (any digits given are ignored).

G - Get the physical file specified from the disk. Care must be exercised that a file is not loaded that will overlay the debugger (locations 0-01377 and 06000-07377). If the file does not exist or contains a record of illegal loader format, a beep will be given. The first digit of the last four entered is the logical drive number from which the file is to be loaded. The following three digits are the physical file number. For example, 02003G will load SYSTEM3/SYS from drive two. To load PFN 0115 from drive 0, simply enter 115G.

I - Increment the current address (any digits given are ignored).

J - Execute a jump to the number given or, if no number is given, to the current address. The alpha machine state is loaded from the values saved in the debugger before the jump is executed.

M - Modify the contents of the current address. The least significant eight bits of the octal number given before the command character are used for the new memory value. If no digits are given, a zero is assumed.

P - Turn on the P-counter display (to the left of the current address). This display is a foreground driven routine which takes the value of the P-counter when the interrupt occurred and displays it vertically. This implies that the value shown is the background P-counter at 32 millisecond sample points. When the display is active, simultaneous depression of the KEYBOARD and DISPLAY keys will cause the debugger to be entered regardless of what is currently being executed in the background. When such entry occurs, the current address points to the location where the background program was interrupted so that execution can be resumed with the J command.

R - Display the saved alpha mode register value. The registers are referenced by number (0-A, 1-B, 2-C, 3-D, 4-E, 5-H, 6-L, and 7-Conditions). The condition code is stored with bits 7=Carry, 6=Sign, bits 5 through 2 always zero, 1=(-Zero and -Sign), and 0=(-Zero and -Parity). (The easiest way to understand this is to realize that the condition code as

displayed, added to itself, results in restoring all four conditions to their entry values.) When a register is displayed, the address shown is the memory location used to store the value of that register. This does not, however, affect the current address. The registers may be initialized for a C or J command by simply storing into the memory locations displayed when the registers are displayed.

X - Turn off the P-counter display.

- Clear all break points. The current address will reflect the location of the last point cleared.

. - Perform the M command followed by the I command.

CANCEL - Erase the entered number without changing the current address.

ENTER - Change the current address to the digits entered. If no digits are entered, the current address in effect will be displayed.

40.7.2 KEYIN$ - Obtain a Line from the Keyboard

KEYIN$ obtains a string of characters from the keyboard, displaying them on the screen and storing them in memory as they are entered. Its operation is identical to the KEYIN$ routine contained in the Cassette Tape Operating System. When KEYIN$ is called, the cursor is turned on and characters requested. Backspacing off the beginning of the line, entering more than the specified maximum number of characters, or running off the screen is prevented. The routine turns off the cursor and returns when the ENTER key is depressed.

Entry point: 01157

Parameters:
C = maximum number of characters accepted
D = initial horizontal cursor position
E = vertical cursor position
HL= starting location of input buffer

Exit conditions:
String terminated by 015
HL= pointing to the 015
D = horizontal position of ENTER
E = unchanged

40.7.3 DSPLY$ - Display a Line on the Screen

DSPLY$ displays a string of characters stored in memory on the screen. Certain characters denote control functions according to the following table:

003 - end of string
011 - new horizontal position follows
013 - new vertical position follows
015 - end of string with CR/LF
021 - erase to end of frame
022 - erase to end of line
023 - roll up one line

This routine is identical in function to the DSPLY$ routine in the Cassette Tape Operating System. If the string to be displayed starts with either or both horizontal or vertical cursor controls, then either or both of the corresponding values need not be in D or E at entry. If the cursor is not positioned on the screen with DE OR 011 and 013 the results of 021, 022, or 023 are undefined.

Entry point: 01162

Parameters: D = initial horizontal cursor position
E = initial vertical cursor position
HL points to string in memory

Exit conditions: DE = cursor position after the last character displayed
HL = byte after the string terminator

40.8 DOS FUNCTION Facility (DOSFNC)

The page of memory located between 07400 and 07777 contains a special loader and overlay area. This "loader" can load any one of up to 255 DOS overlays, each up to 124 bytes long. The loader resides in the first half of the page and the overlays all load into the second half of the same page. The overlays reside on disk in physical file 7, called SYSTEM7/SYS. The design of the DOS FUNCTION loader is such that overlays are loaded only if necessary; i.e. if the same overlay is called several times in sequence, it is not reloaded each time. The overlays provide the DOS assembly language programmer with many useful utility functions. Parameterization of DOS FUNCTIONs varies with the individual functions, the only basic requirement being that on entry to the DOS FUNCTION loader, the A register contains the function number (1-255). Use of functions not yet installed will

produce indeterminate results, but may result in format traps, range traps, processor halts, and the like. DOS FUNCTIONs are normally loaded from the SYSTEM7/SYS on drive zero.

Upon the first call to DOSFNC (the DOS FUNCTION loader), SYSTEM7/SYS is opened as LF0 and the LFT entry saved internally to the DOS FUNCTION loader. Upon subsequent calls to DOSFNC, the entry is simply moved back into the LFT, eliminating the need to re-open SYSTEM7/SYS each time a function is loaded. The file is only closed by the reloading of the DOS, either by depressing RESTART or by a program passing control to BOOT$, EXIT$, or ERROR$.

Since new DOS functions will be added as necessary the following descriptions should not be considered exhaustive.

Entry point: 07400

Parameters: A = Function number (1-0377)
Others required by individual functions

Exit conditions: Defined separately for each function

40.8.1 FUNC-1 Retrieve Directory and C.A.T. Addresses

Uniform attributes for all subfunctions:

On entry,	A = function number (1) C = subfunction number (0,1,2,3,4,5,6,7)
On exit,	B,C,H,L all unchanged CARRY FALSE: function completed successfully CARRY TRUE: invalid subfunction number

all other entry/exit parameters and conditions are described separately for each individual subfunction.

DOS FUNCTION: 1 SUBFUNCTION: 0

return the address of a specified directory sector in DE

On entry,	B = directory sector number (0-15) OR PFN of entry in the directory sector
On exit,	A indeterminate DE = PDA of specified directory sector

DOS FUNCTION: 1 SUBFUNCTION: 1

return the two byte physical disk address for each of the 16 prime directory sectors, into a 32-byte work area provided by the user.

On entry,	HL => 32-byte work area to receive the PDAs
On exit,	all registers restored user-provided work area contains 16 PDAs, one corresponding to each prime directory sector, in ascending order.

DOS FUNCTION: 1 SUBFUNCTION: 2

return the two-byte physical disk address of each of the 16 directory sector backups, in ascending order, into a 32-byte user-provided work area.

On entry,	HL => 32-byte area to receive the 16 PDAs
On exit,	all registers restored

work area contains 16 PDAs (LSB,MSB)

DOS FUNCTION: 1 SUBFUNCTION: 3

return the physical disk address of the prime cluster allocation table (CAT) in the DE register pair.

On exit, A indeterminate
DE = PDA of prime CAT

DOS FUNCTION: 1 SUBFUNCTION: 4

return the physical disk address of the backup CAT

On exit, A indeterminate
DE = PDA of backup CAT

DOS FUNCTION: 1 SUBFUNCTION: 5

return the physical disk address of the lockout CAT

On exit, A indeterminate
DE = PDA of lockout CAT

DOS FUNCTION: 1 SUBFUNCTION: 6

return the physical disk address of the lockout CAT backup

On exit, A indeterminate
DE = PDA of lockout CAT backup

DOS FUNCTION: 1 SUBFUNCTION: 7

return the address of a backup directory sector (in DE)

On entry, B = backup directory sector number (0-15)
OR PFN of a file entry contained therein
On exit, A indeterminate
DE = PDA of backup directory sector

40.8.2 FUNC-2 Retrieve Directory Sector or Filename

Uniform attributes for all subfunctions:

```
On entry,       A = function number (2)
                C = subfunction number (0,1,2)
On exit,        ALL REGISTERS RESTORED
                CARRY TRUE implies error or invalid subfunction
                number
```

all other entry/exit parameters and conditions are described separately for each individual subfunction.

DOS FUNCTION: 2 SUBFUNCTION: 0

read in the directory sector containing the 16-byte directory entry corresponding to the PFN given, on a specified logical drive.

```
On entry,       D = PDN (logical drive number of file)
                E = PFN
                B = LFN as per DOS standard; (0, 16, 32, 48)
On exit,        CARRY FALSE:  selected directory sector is in
                        buffer specified, which is the selected
                        buffer upon exit.
                CARRY TRUE:   indicates I/O error.
                  Further defined as follows:
                  ZERO FALSE:  specified drive is off-line
                  ZERO TRUE:   unable to read sector due to CRCC
                        error during read, or unrecoverable
                        failure to find sector
```

DOS FUNCTION: 2 SUBFUNCTION: 1

get the 16-byte directory entry corresponding to a specified PFN on a given logical drive.

```
On entry,       B,D,E set as for subfunction 0.
                HL => 16 byte area to receive the entry
On exit,        CARRY FALSE:  entry is in user's area.
                CARRY TRUE:   as for subfunction 0.
```

DOS FUNCTION: 2 SUBFUNCTION: 2

get name/ext (pfn) for a specified numbered file on a specified logical drive. (Same basic format as used by DOS CAT command).

On entry, B,D,E as for subfunction 0.
HL => 20-byte receiving field.

On exit, CARRY FALSE: user's 20-byte area contains the name, extension and PFN of the specified file, for example:

EDIT/CMD (037)

where the right paren is followed by an 03
UNLESS: ZERO TRUE: implies that the file number specified does not exist.

CARRY TRUE: as for subfunction 0.

NOTICE: the use of THIS SUBFUNCTION ONLY (of those in DOS FUNCTION 2) requires that the DOS command interpreter be present (the command interpreter resides from 013400-017000).

40.8.3 FUNC-3 Retrieve R.I.B. Information

Uniform attributes for all subfunctions:

On entry, A = function number (3)
C = subfunction number (0,1,2,3)

All other entry and exit parameters and conditions are described separately for each individual subfunction.

DOS FUNCTION: 3 SUBFUNCTION: 0

return the number of sectors allocated to a file on disk

On entry, DE = proper OPEN$ parameters defining file to be accessed
B = drive number (like C as provided for OPEN$)

On exit, CARRY FALSE: function completed successfully
HL = length of file (MSB,LSB) in sectors
RIB for file specified is in LF0
disk buffer

CARRY TRUE: indicates an error occurred, any one

of:

OPEN failed on file specified
unable to read RIB;
parity or drive off line.

DOS FUNCTION: 3 SUBFUNCTION: 1

get the RIB for a specified file into the LF0 disk buffer

On entry, all parameters indentical to those for subfunction 0.
On exit, all registers restored
CARRY FALSE: function completed successfully
RIB for file specified is
in LF0 disk buffer
CARRY TRUE: indicates an error return.
Any of the following:
OPEN on specified file failed
RIB unreadable due to parity or drive
off-line

DOS FUNCTION: 3 SUBFUNCTION: 2

read a RIB for a file, given the first two bytes of the directory entry

On entry, D = RIB pointer, (msb) from directory or LFT
E = RIB pointer, (lsb) from directory or LFT
B = drive number
On exit, all registers restored
CARRY FALSE: subfunction completed successfully.
RIB is in the LF0 disk buffer
CARRY TRUE: RIB unreadable due to parity or
drive off line
ZERO FALSE: specified drive off line
ZERO TRUE: parity error during read

DOS FUNCTION: 3 SUBFUNCTION: 3

return segment descriptor information from a RIB

On entry, RIB is in LF0 disk buffer
CARRY TRUE: RIB unreadable due to parity or
drive off line
ZERO FALSE: specified drive off line
ZERO TRUE: parity error during read

DOS FUNCTION: 3 SUBFUNCTION: 3

return segment descriptor information from a RIB

On entry, RIB is in LF0 disk buffer
BUFADR field in LF0 LFT entry points to segment descriptor
On exit, E,H,L, LF0 buffer unchanged.

In addition:

CARRY TRUE: function completed successfully
A = starting cylinder number for segment
B = starting cluster number for segment
C = number of sectors in the segment
BUFADR points to next segment descriptor; RIB undisturbed
CARRY FALSE: implies BUFADR pointed after logical end of RIB
BUFADR contents undefined

40.8.4 FUNC-4 Retrieve DOS Configuration Information

Uniform attributes for all subfunctions:

On entry, A = function number (3)
C = subfunction number (0,n)
On exit, CARRY FALSE: function completed successfully
CARRY TRUE: possibly invalid subfunction number.

Different subfunction numbers return different DOS configuration bytes. These values, returned in A, are numeric items which change in value depending upon which DOS is running. The subfunction numbers, along with the significance of the returned value, are:

00 - LETTER OF THE DOS BEING DEALT WITH
01 - DOS VERSION NUMBER OR LETTER
02 - DOS RELEASE NUMBER OR LETTER
03 - TOTAL NUMBER OF CYLINDERS ON DISK (E.G. 203)
04 - MAXIMUM SUPPORTED LOGICAL DRIVE NUMBER (3,7,15)
07 - CLUSTER MASK (E.G. 0340)
010 - CLUSTER NUMBER INCREMENT

(E.G 040)
011 - SECTOR MASK (E.G. 037)
012 - MAXIMUM SECTOR NUMBER IN PDA
(E.G. 23)
013 - NUMBER OF SECTORS/CLUSTER
(E.G. 3,6,24)
014 - NUMBER OF CLUSTERS/CYLINDER
(E.G. 4,8)

40.8.5 FUNC-5 Request Access to System Tables

Uniform attributes for all subfunctions:

On entry, A = function number (3)
C = subfunction number (0,1)

All other entry and exit parameters and conditions are described separatly for each individual subfunction.

DOS FUNCTION: 5 SUBFUNCTION: 0

request exclusive update permission to system table sectors on disk

On entry, D = physical drive number (PDN) of drive
On exit, CARRY FALSE: function completed successfully exclusive use of specified drive guaranteed
CARRY TRUE: indicates an error occurred.

DOS FUNCTION: 5 SUBFUNCTION: 1

release exclusive update authority for system table sectors on disk

On entry, all parameters indentical to those for subfunction 0.
On exit, same as for subfunction 0.
CARRY FALSE: function completed successfully
CARRY TRUE: indicates an error return.

40.8.6 FUNC-6 Test KEYBOARD and DISPLAY Key Status

Uniform attributes for all subfunctions:

```
On entry,       A = function number (6)
                C = subfunction number (0)
```

DOS FUNCTION: 6 SUBFUNCTION: 0

Check the status of the KEYBOARD and DISPLAY keys
If DISPLAY key is pressed, WAIT$ is called by the function before returning.

```
On entry,       Doesn't matter
On exit,        CARRY TRUE if illegal subfunction
                SIGN TRUE:  KEYBOARD key pressed
                PARITY TRUE:  DISPLAY key pressed
                All registers restored
```

40.8.7 FUNC-7 Test the Disk Buffer Memory

Disk buffer memory test function

This DOS FUNCTION performs a rotating, cycling test of the disk controller buffer memories. It returns upon the keyboard becoming READ READY, or upon encountering a buffer failure, whichever occurs first.

```
On entry,       Doesn't matter
On exit,        all registers unchanged
                ZERO TRUE = buffer memory test completed normally
                ZERO FALSE = failure indicated in buffer memories
```

40.8.8 FUNC-8 Timed Pause

Pause function

This DOS FUNCTION provides the user program with a timed pause. The requested pause may be up to over four hours long.

```
On entry,       B = foreground process number to use (0-7)
                CDE = number of milliseconds to pause
                C = most significant byte
                E = least significant byte

On exit,        All registers unchanged
```

Note that if foreground process numbers 4-7 are used, the wait

time is effectively multiplied by four, allowing a maximum wait time in excess of eighteen hours. Also note that the time required to start up the DOS FUNCTION is not considered part of the time paused. Since the DOS FUNCTION may or may not be resident when called, this function may wait longer than the quantity in CDE and therefore must not be used for timing really critical, short term intervals.

40.8.9 FUNC-11 RAM Screen Loader

Uniform attributes for all subfunctions:

On entry, A = function number (11)
C = subfunction number (0,1,2)

All other entry and exit parameters and conditions are described separately for each individual subfunction.

DOS FUNCTION: 11 SUBFUNCTION: 0

Load one or more character combinations into the RAM display character generator.

On entry, B = default first character to be loaded
HL = starting address of character set definition list

The list consists of consecutive entries of either five or six bytes each. The first byte, if present, indicates the 7-bit character combination whose bit pattern definition follows. The presence of the first byte is indicated by its sign bit being set. If the first byte of the first entry is not present, the 7-bit character combination in the B register is used instead. The definition list may contain any mixture of six byte and five byte entries. The end of the list is indicated by an 0200. This implies that the bit combination displayed for a binary zero cannot be imbedded in a list, but can only appear at its beginning; null lists are not allowed. The five data bytes following represent the five columns of bits for each displayed character and can each have values of 0 (a blank column) to 0177 (a vertical line). The 0100 bit is at the top of the character displayed; the 1 bit is on the bottom row of the displayed character.

On exit, CARRY FALSE, ZERO FALSE implies RAM display not

present
CARRY FALSE, ZERO TRUE indicates normal completion
CARRY TRUE indicates error (should not occur)

DOS FUNCTION: 11 SUBFUNCTION: 1
Load a single character combination to RAM display

On entry, B = default character to be loaded
HL = address of five or six byte bit pattern definition

The first byte, if present, takes precedence over the character indicated by the B register. Presence of the first byte is indicated by the sign bit being set.

On exit, CARRY FALSE, ZERO FALSE implies RAM display not present
CARRY FALSE, ZERO TRUE indicates normal completion

DOS FUNCTION: 11 SUBFUNCTION: 2

Subfunction two requests reloading of the standard character set on program termination. Calling this subfunction will result in the standard DOS character set being reloaded upon the next entry to DOS$. Entry to the DOS at DOS$ is the result of transfer of control to EXIT$, BOOT$, ERROR$ as well as DOS$. Return to the DOS via NXTCMD, CMDAGN, and CMDINT do not result in the display being immediately reloaded, (but it still will be upon subsequent entry at DOS$ as described).

40.9 Cassette Handling Routines

Standard record formats, identifiers, and file marker record conventions on cassettes are established by the Cassette Tape Operating System. Routines capable of dealing with cassettes in a manner compatible with CTOS are provided as part of the Disk Operating System to enhance its overall capability. For detailed information on cassette format and organization, see the Cassette Tape Operating System Manual.

All of the DOS cassette routines are foreground driven and, with the debugging facility, are the only routines within the system which make use of the foreground handling facility. Being foreground driven, however, does not alter the way with which the routines are dealt since all interfacing between the background

and foreground is handled by the system. It does allow increased speed of operation with the cassettes since the user may be processing one record while the next is being read from or written to the tape. This is evident in the way the DOS slews the tape when transferring information between it and the disk.

Some of the cassette handling routines initiate foreground action and then return immediately to the user while others wait for I/O completion. All of the routines wait for any uncompleted I/O to finish before starting something new. Note that in the cases of reading or writing on the same deck, requesting the next operation before the completion of the first will cause the tape to automatically slew instead of stopping between records. This is only in the case of a read followed by another read or a write followed by another write on the same deck. The only cases where caution must be exercised is in the read and write routines which return immediately after starting the I/O operation. If the user does not wait for the transfer to complete, he could try to use the data before it is read or change the data before it is written. In the second case, records with incorrect parity will usually be generated. Routines are provided, however, which automatically wait for the transfer to complete, relieving the user of having to concern himself with the fact that the routines are foreground driven if he has no need for the advantages.

The various error conditions associated with cassette handling can be trapped by the user. If the trap is not set, an error message similar to the error message generated by CTOS is displayed and the DOS reloaded. If the trap has been set, the address specified will be jumped to and the trap cleared. The traps are identified in the error message by a letter similar to the CTOS identification. In the relevant cases, the same letter is used in the DOS as is used in the CTOS. In the following routine descriptions the relevant letter will be given in the 'Traps' section.

Most of the cassette routines are parameterized by a deck number given in the B register. This number is a zero for the rear deck and a one for the front deck.

40.9.1 TPBOF$ - Position to the Beginning of a File

TPBOF$ positions the cassette in the specified deck to the specified file. The search for the file marker of the desired file is started with backward motion of the tape. If a marker of lower value than the file number requested or the beginning of the tape is encountered, the search will be reversed to the forward motion of the tape. If then a marker of larger value than the

file number requested, the end of the tape, or a record of unrecognizable format is encountered, an error G will be given. Otherwise, the file is left positioned before the first data record.

Entry point: 010000

Parameters: B = deck number
C = physical file number (0-0177)

Exit conditions: none

Traps:

D	unrecognizable record found
G	file could not be found

40.9.2 TPEOF$ - Position to the End of a File

TPEOF$ moves the tape forward until the next file mark is found. It then backspaces the tape one record to leave it at the end of the current file.

Entry point: 010005

Parameters: B = deck number

Exit conditions: none

Traps:

D	unrecognizable record found
E	end of tape encountered

40.9.3 TRW$ - Physically Rewind a Cassette

TRW$ rewinds the cassette on the selected deck by first slewing backwards to ensure that the tape is not on the trailer and then performing a hardware rewind.

Entry point: 010012

Parameters: B = deck number

Exit conditions: none

Traps: none

40.9.4 TBSP$ - Physically Backspace One

TBSP$ simply executes a hardware backspace function. No checking is performed on the data passed over. However, backspacing onto clear leader causes an end of tape trap.

Entry point: 010017

Parameters: B = deck number

Exit conditions: none

Traps: E beginning of tape encountered

40.9.5 TWBLK$ - Write an Unformatted Block

TWBLK$ writes the specified number of bytes (0-255; 0 causes 256 to be written) from the memory buffer specified onto the cassette in the deck specified. Only the bytes specified will be written on the tape.

Entry point: 010024

Parameters: B = deck number
C = number of bytes to write (0 for 256)
HL points to start of buffer

Exit conditions: none

Traps: E end of tape encountered
Z premature deck ready status

40.9.6 TR$ - Read a Numeric CTOS Record

TR$ reads a record of CTOS numeric format into the memory locations specified. The length of the record is stored in the specified memory location and the data bytes are stored in the locations that follow. Return is made from TR$ as soon as the read operation is started but the user cannot use the data until the operation has been completed (see TCHK$). One way to check for operation completion is to call TR$ again with a different buffer as its parameter. Return from the second call will be made as soon as the first operation is completed. This is the mechanism via which multiple buffering is normally achieved. Note that tape motion will not cease if TR$ is called within five milliseconds of the end of the previous record.

If parity problems arise, TR$ tries up to 5 times to read the tape before giving a parity failure trap. Other traps given are end of tape and end of file. If an end of file trap is given, the tape is positioned before the file marker.

Entry point: 010031

Parameters: B = deck number
HL points to data storage location

Exit conditions: none

Traps:

D	parity failure
E	end of tape encountered
F	end of file encountered

40.9.7 TREAD$ - TR$ and Wait for the Last Character

TREAD$ performs the TR$ function and then waits for the last character to be read from the tape. This routine should be used when multiple buffering is not being performed since it relieves the user from having to explicitly wait for the last character to be read.

Entry point: 010034

Parameters: same as for TR$

Exit conditions: none

Traps: same as for TR$

40.9.8 TW$ - Write a Numeric CTOS Record

TW$ writes the specified memory locations in a record of standard CTOS numeric format. It uses (for parity generation) the three locations preceeding the memory location specified which contains the number of bytes to be written and is followed by that number of data bytes.

TW$ returns as soon as the write operation is started. The user must be careful not to change any of the memory locations given as parameters before the last byte has been transferred. This can be achieved by either calling TCHK$ and waiting for completion status or calling TW$ with the next buffer if multiple buffering is being used. Note that tape motion will not cease if TW$ is called before the middle of the IRG is reached from the previous write (140 milliseconds after the last character is

written when using a 7.5 ips deck).

Entry point: 010037

Parameters: same as for TR$

Exit conditions: none

Traps:

E	end of tape encountered
Z	premature deck ready status

40.9.9 TWRIT$ - TW$ and Wait for the Last Character

TWRIT$ executes the TW$ routine and then waits for the last byte to be written on the tape. This routine should be used when multiple buffering is not being performed since it relieves the user from having to explicitly wait for the last byte to be written.

Entry point: 010041

Parameters: same as for TR$

Exit conditions: none

Traps: same as for TW$

40.9.10 TFMR$ - Read the Next File Marker

TFMR$ reads the tape until a file marker record is found. A trap occurs if a record is encountered that is neither a file marker nor a CTOS numeric data record.

Entry point: 010045

Parameters: B = deck number

Exit conditions: C = PFN of marker found
Tape positioned after marker record

Traps:

D	unrecognizable record found
E	end of tape encountered

40.9.11 TFMW$ - Write a File Marker Record

TFMW$ writes a file marker record that contains the number specified.

Entry point: 010050

Parameters: B = deck number
C = PFN to be written

Exit conditions: none

Traps:

E	end of tape encountered
Z	premature deck ready status

40.9.12 TTRAP$ - Set an Error Condition Trap

TTRAP$ allows the user to trap the various errors associated with cassette I/O. If the trap is not set, an error message of the form

*** ERROR X ON DECK Y ***

will be displayed, where X is one of the letters shown below and Y is a 1 for the rear deck and a 2 for the front deck. The trap is specified by a number according to the following table:

3 - D - parity error
4 - E - end of tape
5 - F - end of file
6 - G - unfindable file

In addition, error Z (cannot be trapped) indicates that the deck ready status bit came true while a record was being written. This implies that the write routine fell behind in writing characters and most probably indicates that the foreground interrupt handling was disrupted in some fashion (interrupts were disabled too long or an interrupt driven routine was running which imposed too much overhead). It may also caused by the tape being write protected (left rear tab punch out).

Traps can be cleared by setting their addresses to zero. When the event which causes a trap occurs, that trap is cleared and control passed to the address indicated with the deck number in the B register (0 for rear and 1 for front deck).

Entry point: 010053

Parameters: C = trap number (above)
DE= trap address (0 clears trap)

Exit conditions: none

Traps: none

40.9.13 TWAIT$ - Wait for I/O Completion

TWAIT$ waits for any tape operation active to complete. This does not mean that physical motion has stopped since TR$ and TW$ indicate I/O completion when the last character has been transferred. It does mean that all data is free to be processed by the user. TWAIT$ also executes any traps pending upon the completion status being set.

Entry point: 010056

Parameters: none

Exit conditions: B, C, D, and E registers preserved

Traps: any trap pending will be executed

40.9.14 TCHK$ - Get I/O Status

TCHK$ sets the tape demand flag in the carry condition flag and loads the tape handling status in the A register. The handling status codes are as follows:

000 - PBOF in progress
002 - PEOF in progress
004 - Rewind in progress
006 - Record read in progress
010 - Backspace in progress
012 - File mark read in progress
014 - Record write in progress

377 - Normal completion
206 - Parity error
210 - End of tape
212 - End of file
214 - File not found
262 - Premature deck ready status

Normal use of the cassette routines will not require the user to deal with these status codes or even use the TCHK$ routine. They are provided here to facilitate understanding the listing of the

routines.

Entry point: 010061

Parameters: none

Exit conditions: Carry condition = demand flag
A = status code (above)

Traps: none

40.10 Command Interpreter Routines

This section deals with a series of user-available routines available within the command interpreter. Note that these routines are only available for use if the user program does not overlay the command interpreter, which resides in locations 012400-016777.

The first four of these entry points are really more like "exit points", since they are places in the DOS to which users may return in place of EXIT$. The primary advantage to using them in place of EXIT$ is that none of these four entry points result in the DOS being reloaded, a process which takes significant time. Note that since they do not reload the DOS, programs which exit through CMDINT, DOS$, CMDAGN, or NXTCMD must not have overstored any part of the DOS; i.e. they should run completely in locations 017000 upwards. Also, these "exit points" do not clear any traps that the user may have set; therefore the user should clear any traps he has set before exiting in this manner. If this is not done, the system will most likely go astray upon the first subsequent occurrence of a trapped situation.

Most of the other routines documented in this section are routines which are used by one or more of the DOS command programs supplied either on the DOS Generation or DOS Utilities tapes. Since these routines are pointed to by the command interpreter's entry point table and are used by some of the DOS commands, they are documented here primarily for the sake of completeness; not to suggest that every DOS programmer will find them wonderfully useful for his particular application.

40.10.1 CMDINT - Return & Scan MCR$ line

CMDINT closes files 1-3 if necessary and processes MCR$ just as it would a command line entered by an operator at the keyboard. (This results in executing the program indicated by the command line.)

Entry point: 01165

Parameters: MCR$ (an 80 byte area of memory starting at 01400) should contain a string resembling a command line terminated with a 015.

Exit conditions: Does not return

40.10.2 DOS$ - Return & Display Sign On

DOS$ first loads the RAM screen, if there is one, with the character set contained in SYSTEM6/SYS (or CHARSET/SYS if it exists). Once the RAM display has been loaded, it is not loaded until either another bootstrap from cassette, or the appropiate DOS function is invoked by a DOS program. DOS$ then causes a program which has been AUTO'd to be executed. If no programs are set for auto-execution, the DOS sign-on is displayed, files 1-3 are closed if necessary, and the familiar "READY" message displayed. Note again that any traps set by the user program (e.g. via TRAP$) are not cleared unless the DOS is reloaded. This implies that if a user program sets any of the traps and wishes to return via DOS$, NXTCMD, or CMDAGN, it must first clear any traps it has set to prevent the DOS from going astray. DOS$ is the normal starting point of the DOS when a bootstrap operation or a jump to BOOT$, EXIT$, or ERROR$ occurs.

Entry point: 013400

Parameters: none

Exit conditions: Does not return

40.10.3 NXTCMD - Return & Say "READY"

NXTCMD causes files 1-3 to be closed and displays the familiar DOS "READY" message.

Entry point: 013403

Parameters: none

Exit conditions: Does not return

40.10.4 CMDAGN - Return & Give Message

CMDAGN causes files 1-3 to be closed and displays a user-supplied message before returning to the command interpreter.

Entry point: 013406

Parameters: HL = address of DSPLY$-format string
DE unused; string should position cursor

Exit conditions: Does not return
DOS CHAIN facility aborts if active

40.10.5 GETSYM - Get Next Symbol from MCR$

GETSYM causes the next sequential symbol in MCR$ to be scanned off and stored in an 8-byte field called SYMBOL located at 013472. The starting byte scanned in MCR$ is pointed to by INPTR, a byte at location 013455. (INPTR is the LSB of the current byte in MCR$.) The symbol (leading spaces are ignored) must contain only upper case alphabetic or numeric characters. The first illegal character encountered terminates the scan; the illegal, terminating character is stored for the user's inspection (at SYMBOL+8) and SYMBOL is padded on the right with spaces if necessary. If the symbol is longer than eight characters, the first eight only are used; remaining characters, through the terminator, are scanned but not stored. (The terminator is stored at SYMBOL+8 in any case.) On exit, INPTR points after the terminating character unless the terminator is an 015 or a semicolon, in which case INPTR points to the terminator.

Entry point: 013411

Parameters: INPTR => current byte in MCR$, LSB

Exit conditions: SYMBOL = 8-byte symbol as described above
A, SYMBOL+8 = terminator character
INPTR => byte after symbol terminator in MCR$
(except as noted above)
All other registers indeterminate

40.10.6 GETCH - Get the Next Character from MCR$

GETCH obtains the next character from the Monitor Communication Region (MCR$) and returns it in A. The address of the character to be returned is obtained by using the most significant byte of the address of MCR$ (which is contained within one page) and the contents of INPTR (location 013455) as the LSB. On exit, if zero is true, A = 015 or a semicolon, and INPTR is not incremented (INPTR is never bumped past an 015 or a semicolon); if zero is false, A is not an 015 or a semicolon and INPTR is incremented.

Entry point: 013414

Parameters: INPTR = LSB of address of byte (see above)

Exit conditions: A = character from MCR$
ZERO TRUE/FALSE as described above
B = entry value of INPTR
C,D,E unchanged

40.10.7 GETAEN - Get Auto-Execute Physical File Number

GETAEN returns the physical file number of the file (on the logical drive specified in C) which is set to be auto-executed by the DOS.

Entry point: 013417

Parameters: C = Logical Drive

Exit conditions: Carry true if I/O error reading the CAT
otherwise, A = auto-execute PFN (0=none)
Zero true if a-e PFN not set
Zero false if A is valid a-e PFN
All other registers indeterminate

40.10.8 PUTAEN - Set or Clear a File to be Auto-Executed

PUTAEN either sets or clears the auto-execute PFN stored in the CAT on the disk in the logical drive specified in C. The change becomes effective upon the next time DOS is entered at DOS$, either by depressing the RESTART key, the auto-restart tab being punched out of the rear cassette and the processor halted, or jumping to EXIT$, ERROR$, BOOT$, or DOS$.

Entry point: 013422

Parameters: A = PFN to be auto-executed (0 to clear)
C = Logical Drive

Exit conditions: All registers indeterminate
Carry true if I/O error updating CAT

40.10.9 GETLFB - Open the User-Specified Data File

GETLFB opens logical file specified in B using the file name, extension, and drive select code, stored in the indicated LFT entry, in the normalized form described in the section on the Command Interpreter. The extension, if blank, is assumed to be "ABS". Note: The logical drive specification field is ignored, since the drive select code field is used instead. If an error occurs, carry is true on return and HL points to a DSPLY$ format string complete with cursor positioning bytes and one of the following messages:

NAME REQUIRED. (first byte of name field is blank)
INVALID DEVICE. (select code = 0376; :DRn wrong)
NO SUCH NAME. (file not found; the file must exist)

Each of the above messages is preceeded by control bytes: 011,0,013,11,023 and followed by an 015. If carry is false upon return, the file named has been successfully opened as logical file one.

Entry point: 013425
B = LFN

Parameters: In LFT specified by LFN; see above

Exit conditions: Carry false if file successfully opened
All registers indeterminate
Carry true and HL => message if OPEN failed

40.10.10 PUTCHX - Store the Character in "A"

PUTCHX stores the A register at the memory location pointed to by HL, increments HL, and decrements a byte counter maintained in E.

Entry point: 013433
HL = address where A is to be stored

Parameters: A = byte to be stored at HL
E = count to be decremented

Exit conditions: B,C,D unchanged
E = entry value - 1
HL = entry value + 1

40.10.11 PUTCH - Alternate Version of PUTCHX

PUTCH is like PUTCHX except it starts by setting the most significant bit of A to zero and that if A then contains a space (040) it immediately returns zero true; in which case A is not stored, HL not incremented, and E not decremented.

Entry point: 013430

Parameters: same as PUTCHX

Exit conditions: same as PUTCHX except as described above

40.10.12 PUTNAM - Format a Filename from Directory

PUTNAM is a routine which extracts a name, extension and physical file number for a directory entry and puts them into a place in the command interpreter called "NAME" (located at 013513; the field is 19 bytes long and followed by an 03.) Since this routine is used by the CAT command, the format of the names produced by PUTNAM should be familiar to all DOS users.

Note that on entry, only the <u>most</u> significant 4 bits of C are used, and that CURLOC (location 013463) is to contain the two-byte PDA of the directory sector (LSB,MSB).

Entry point: 013436

Parameters: the directory sector in the disk buffer
B = LFN indicating which buffer
C = PFN of entry being extracted
CURLOC = PDA of directory sector

Exit conditions: CURLOC unchanged
disk buffer unchanged
B unchanged
all other registers indeterminate
ZERO TRUE: file does not exist

40.10.13 MOVSYM - Obtain the Symbol Scanned by GETSYM

MOVSYM moves the eight-byte SYMBOL described in the section on GETSYM into the eight-byte area pointed to by DE.

Entry point: 013441

Parameters: D,E = address of user's eight-byte area

Exit conditions: B unchanged. All other registers indeterminate.

40.10.14 GETDBA - Obtain Disk Controller Buffer Address

GETDBA extracts the current disk buffer address in the format acceptable to GETR$ from one of the four LFT entries. It does this by getting the BUFADR from the specified LFT entry and subtracting three from it. On return, H is the address MSB pointing into the command interpreter data area.

Entry point: 013444

Parameters: B = LFN (0,16,32,48)

Exit conditions: A = BUFADR as described above
H as described above
B,C,D,E unchanged

40.10.15 SCANFS - Scan Off File Specification

SCANFS scans a file specification of the form FILENAME/EXT:Drive (as discussed under FILE names) pointed to by HL into a 16 byte area pointed to by DE. THe area pointed to by DE is treated as an LFT entry, that is, the first byte is a drive select code (0376 meaning invalid drive spec, 0377 meaning unspecified drive spec, or the binary drive number), the second byte is 0377 indicating the file is closed, bytes 3 thru 10 are the file name (blank if not given), bytes 11 thru 13 are the extension (blank if not given), and bytes 14 thru 16 are the normalized drive spec (blank if not given). The scanned drive spec may be 2 to 7 characters long; the first character must be "D", the second may be "R", and the remaining must be digits. Therefore ":D0" and ":DR00014" are both legal representations. The normalized represention consists of a "D" followed by "h" and the single digit given or "D" followed by the two digits given; for instance, the above examples in normalized form would be "DR0" and "D14" respectively. The scan is terminated by any

non-alphanumeric character other than ":" or "/".

Entry Point: 013447

Parameters: DE => "LFT TABLE" entry
HL => string to be scanned

Exit Conditions: DE => byte following "LFT TABLE" entry
HL => byte after terminator (unless 015 or ";"
in which case it points to terminator)

40.10.16 TCWAIT - Test controller memory & wait

TCWAIT is the point in the COMMAND INTERPRETER where it loops testing the disk controller buffer memory while waiting for a command to be keyed in. It is used primarily by the CHAIN command to trap programs returning to D.O.S.

Entry Point: 013452

Parameters: none

Exit Condition: does not return

40.11 User Supported Input/Output

When the user desires to use I/O devices other than the keyboard, display, disk, or cassettes, he will use a routine that is not part of the operating system. Many of these devices (for instance, the communications channel) will be serviced by foreground processes which run with interrupts disabled. However, if the user does access an I/O device from a background process, he must realize that as long as interrupts are enabled, some other device can be addressed by a foreground routine. For this reason, the user must disable interrupts between the time he addresses his device and the time he uses it. To reduce the amount of foreground processing real time jitter (discussed earlier) as much as possible, the aim in writing background I/O routines should be to minimize the amount of time that interrupts are disabled. This implies that devices accessed from background programs must be addressed every time they are used. For example:

```
GETBYT  EI              Enable interrupts in case
        LA    DEVADR      looping
        DI              Disable interrupts
        EX    ADR       Address the device
```

```
IN              Get the device status
ND    2         Check for required bits
JTZ   GETBYT    Wait if not set
EX    DATA      Else get the byte
EI              Enable interrupts after
IN                 the data input
RET
```

Note that a little cheating on time was done in the interest of program length. Since the INPUT in DATA mode was done without enabling interrupts, re-disabling them and re-addressing the device was not necessary. One should be judicious in the trade off employed in exercising this freedom.

Note: The user must not do I/O to the disk controller from foreground-driven routines or results can be unpredictable. The DOS disk drivers allow user foreground routines to get control in the midst of a disk I/O operation, under the assumption that the foreground routine will not do anything to the disk controller which would confuse it.

CHAPTER 41. ERROR MESSAGES

PARITY FAILURE DURING READ
: A parity fault occurred while a disk data record was being read.

PARITY FAILURE DURING WRITE
: A parity fault occurred while a disk data record was being written.

RECORD FORMAT ERROR
: The physical file number or logical record number in the record read did not match the values contained in the logical file table.

RECORD NUMBER OUT OF RANGE
: The record accessed had a logical record number less than zero or, during reads, was outside the physical space allocated to the file.

WRITE PROTECT VIOLATION
: An attempt was made to write on a file that had its write protection bit set.

DELETE PROTECT VIOLATION
: An attempt was made to delete a file that had either its write or delete protection bit set.

FILE SPACE FULL
: An attempt was made to allocate space when either the disk was physically full or no more segment descriptor slots were available in the RIB for the given file.

DRIVE OFF LINE
: The drive went off line after the file was opened.

LOGICAL FILE NOT OPEN
: An attempt was made to use an entry in the logical file table that was not opened for use with some file.

INVALID LOGICAL FILE NUMBER
: A routine was called with the logical file number parameter not zero through three.

INVALID DRIVE NUMBER
A routine was called with the drive number not zero through the defined drive number limit (or 0377, if allowed).

INVALID TRAP NUMBER
The TRAP$ routine was called with a trap number not between zero and seven.

FAILURE IN SYSTEM DATA
An unrecoverable parity error occurred while the system was dealing with one of the disk tables or a retrieval information block, or a RIB with incorrect format was accessed.

INVALID PHYSICAL FILE NUMBER
A physical file number reserved for the system was illegally referenced.

INTERNAL SYSTEM ERROR
The error message routine was parameterized with an invalid error message number!

ERROR X ON DECK Y
A cassette routine error has occurred. The X indicates the type of error according to the following table:

D - parity error
E - end of tape
F - end of file
G - unfindable file
Z - write failure

CHAPTER 42. ROUTINE ENTRY POINTS

These entry points are contained in a file called DOS/EPT.

Loader Routines

01000	BOOT$	reload the operating system
01003	RUNX$	load and run a file by number
01006	LOADX$	load a file by number
01047	GETNCH	get the next disk buffer byte
01052	DR$	read a sector into the disk buffer
01055	DW$	write a sector from the disk buffer
01060	DSKWAT	wait for disk ready

Foreground Routines

01033	CS$	change process state
01036	TP$	terminate process
01041	SETI$	initiate foreground process
01044	CLRI$	terminate foreground process

File Handling Routines

Symbolic File Referencing

01063	PREP$	open or create a file
01066	OPEN$	open an existing file
01071	LOAD$	load a file by name
01074	RUN$	load and run a file by name

Logical File Referencing

01077	CLOSE$	close a file
01102	CHOP$	delete space in a file
01105	PROTE$	change the protection on a file
01110	POSIT$	position to a record within a file
01113	READ$	read a record into the buffer
01116	WRITE$	write a record from the buffer
01121	GET$	get the next buffer character
01124	GETR$	get an indexed buffer character
01127	PUT$	store into the next buffer position
01132	PUTR$	store into an indexed buffer position
01135	BSP$	backspace one record

Non-file referencing

01011	INCHL	increment HL
01022	DECHL	decrement HL
01140	ERROR$	close all files, exit chain, and reload DOS
01143	BLKTFR	transfer a block of memory
01146	TRAP$	set a disk error condition trap
01151	EXIT$	reload the operating system
01170	WAIT$	DOS wait-a-while "NOP" routine
07400	DOSFNC	DOS function loader

Keyboard and Display Routines

01154	DEBUG$	enter the debugging tool
01157	KEYIN$	obtain a line from the keyboard
01162	DSPLY$	display a line on the screen

Cassette Handling Routines

010000	TPBOF$	position to the beginning of a file
010005	TPEOF$	position to the end of a file
010012	TRW$	physically rewind a cassette
010017	TBSP$	physically backspace one record
010024	TWBLK$	write an unformatted block
010031	TR$	read a numeric CTOS record
010034	TREAD$	TR$ and wait for last character
010037	TW$	write a numeric CTOS record
010042	TWRIT$	TW$ and wait for last character
010045	TFMR$	read the next file marker record
010050	TFMW$	write a file marker record
010053	TTRAP$	set a cassette error trap
010056	TWAIT$	wait for I/O completion
010061	TCHK$	get I/O status

COMMAND INTERPRETER UTILITY ROUTINES

01165	CMDINT	return to command interpreter & scan MCR$ line
013400	DOS&	return to command interpreter & display sign on
013403	NXTCMD	return to command interpreter & say "READY"
013406	CMDAGN	return to command interpreter & give message
013411	GETSYM	get the next symbol from MCR$
013414	GETCH	get the next character from MCR$
013417	GETAEN	get the auto execute PFN
013422	PUTAEN	set the auto execute DFN
013425	GETLFB	open the user-specified file (LFN in B)
013430	PUTCH	store the nonblank character in the A register
013433	PUTCHX	store the character in the A register
013436	PUTNAM	format a filename from a directory block

013441	MOVSYM	obtain the symbol scanned off by GETSYM
013444	GETDBA	obtain the disk controller buffer address
013447	SCANFS	scan off a file specification
013452	TCWAIT	test controller memory and wait for command

CHAPTER 43. DOS QUESTIONS AND ANSWERS

Q. When I write my program, where should I place it in memory?
A. The best address to specify in your SET statement in an assembly language program is 017000. This allows your program full access to the routines in the DOS command interpreter and allows your program to return to the DOS through the NXTCMD and CMDAGN entry points. If the 8.5 K remaining above 017000 is inadequate for your program's needs, you could perhaps start your program at 010000 (assuming your program will not be using the DOS cassette handling routines or command interperter ROUTINES.)

Q. Where should I put the data areas used by my program, at the beginning or at the end?
A. Experience in programming the Datapoint computers has found that generally it is best to put program data areas before the program itself. One advantage of this approach stems from the fact that programs can often be made shorter if most or all of the most commonly used data items are contained within one page of memory, eliminating the need to reload the H register as often. Since programs typically start on a page boundary, this automatically means that the first 256 bytes of your data area will be in one common page. Another advantage of this approach is that a person reading a program is frequently aided by seeing the program's data area and error messages, etc., before he plunges into the code itself. This placement also reduces the number of forward references the assembler must contend with. But don't forget to specify the entry point on your "END" statement! The default entry point is to the first byte of code generated. Yours wouldn't be the first program to start executing your data area!

Q. When my program gets control from the DOS, do I need to save the registers so I can restore them before returning to it?
A. No. Under the DOS the saving and restoring of the system's registers by user programs is not necessary.

Q. Talking about returning to the DOS, how should my program do that?
A. When a user program finishes, the normal termination is by jumping to EXIT$.

Q. Does it matter if my program returns to the DOS (to EXIT$, NXTCMD, CMDAGN, or whereever) with the stack at a different level than when my program started? In other words, if my program calls several levels down into subroutines and the subroutine jumps to EXIT$, will that mess things all up?
A. No. Since the stack wraps around, the level is always relative and it makes no difference what is in the stack when the user returns control to the DOS.

Q. What is the best way to pass parameters to my subroutines? Is there any official convention for this?
A. There is no "official convention" for parameter passing. However, experience with programming under the DOS suggests that passing parameters in the registers as typified by the DOS file handling routine parameterization is both efficient and convenient to use. The DOS convention that abnormal returns from subroutines are indicated by carry being true on exit (and further information indicated by the zero condition being true or false) also has proven to be a very handy technique, and one which user programs can probably make profitable example of.

Q. Can I update my data files with EDIT?
A. Most data files cannot be EDITed. EDIT produces a space and record compressed output, regardless of input file format. Also, EDIT will segment records longer than 79 bytes into two or more records. Only if your data file is compressed and has 79 byte (or smaller) records can EDIT be used on it. In general, do not EDIT a data file; write an update program.

Q. Whats going on when I run a program and nothing happens; the machine just comes back with READY?
A. This is the system's normal action when it finds an unloadable program. Something - a parity error, a non-object record - made the program unloadable. Try COPYing the program to clear any parity errors. APP the program to test for non-object records. It may be necessary to re-assemble the program or get a new object file from tape or another disk.

If you have had questions that may be helpful to others, please forward them to the Software Development Group, Datapoint Corporation so that they may be considered for use in subsequent releases of the DOS User's Guide.