REPORT NO. DOT-TSC-OST-73-6

DOS-32 USER'S MANUAL

J.P. Carlson

MAY 1973
OPERATIONAL HANDBOOK

TECHNICAL REPORT STANDARD TITLE PAGE

1. Report No. DOT-TSC-OST-73-6	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle DOS-32 USER'S MANUAL		5. Report Date May 1973
		6. Performing Organization Code
7. Author(s) J.P. Carlson		8. Performing Organization Report No. DOT-TSC-OST-73-6
9. Performing Organization Name and Address Transportation Systems Center Data Services Division Kendall Square Cambridge, MA. 02142		10. Work Unit No.
		11. Contract or Grant No. C2274A
12. Sponsoring Agency Name and Address Department of Transportation Office of the Secretary Washington, D.C. 20590		13. Type of Report and Period Covered Operational Handbook
		14. Sponsoring Agency Code
15. Supplementary Notes		

16. Abstract

This manual gives information a user needs to use a Honeywell H-632 Disk Operating System (DOS-32). DOS-32 is a core resident, one user, console-oriented operating system which allows the user to control the computer with no interaction with the control panel. A companion document DOS-32 System Manual, describes the internal workings of DOS-32. DOS-32 consists of an interactive command language, a file system and disk management system, a collection of system programs and an extended FORTRAN library. The DOS-32 I/O system controls the disk, paper tape, line printer, card reader and magnetic tape drives.

The system programs include a FORTRAN compiler, a MACRO assembler, a loader, an interactive editor, an on-line debugging program, and a generalized routine that copies data from one storage medium to another. The extended FORTRAN library includes file accessing routines, peripheral device I/O routines, and utility routines to manipulate words, half-words, characters, hexadecimal digits and bits.

17. Key Words · Operating System · Command Language · File System · Disk Management System · Run Time Library	18. Distribution Statement APPROVED FOR TRANSPORTATION SYSTEMS CENTER ONLY. TRANSMITTAL OF THIS DOCUMENT OUTSIDE THE TSC MUST HAVE PRIOR APPROVAL OF THE TSC-ADP SYSTEMS BRANCH.		
19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages 182	22. Price

Form DOT F 1700.7 (8-69)

FOREWORD

This manual gives information a user needs to use a Honeywell H-632 Disk Operating System (DOS-32). DOS-32 is a core resident, one user, console-oriented operating system which allows the user to control the computer with no interaction with the computer control panel. A comparison document, DOS-32 System Manual, describes the internal workings of DOS-32.

Chapter 1 presents an introduction to the general characteristics of the DOS-32 Operating System - the File System, Multiple Disk Organization, Core Usage and BREAK Key Interrupt features.

Because DOS-32 is an interactive system, one of the first things to do is get a feeling for what it's like to sit at the console and communicate with the system. Chapter 2 presents a session at the console in narrative form. In it a typical problem is introduced and the way to handle it using DOS is described. All commands issued and all system responses are shown. This orients you both to general system characteristics and to specific user commands.

Chapter 3 consists of detailed descriptions of all DOS-32 user commands. The emphasis is to present information needed to take advantage of all capabilities of DOS.

Chapter 4 presents detailed descriptions of system and utility routines available for use by user programs. These include file system, routines, teletype and other peripheral device routines, bit, character and half-word manipulation routines and other routines. Section 4.7 describes how to use the file system, teletype, card reader and line printer through FORTRAN I/O statements.

Ten appendixes provide convenient reference for the user. Appendix A provides a glossary of frequently used terms. Appendix B lists all DOS-32 commands, their format and a brief description of each and serves as a memory jogger for the material presented in Chapter 3. Command abbreviations are presented in Appendix C. Appendix D gives current system bugs along with useful programming hints and should be read before using DOS-32. Appendix E gives instructions on initializing DOS as is presented at the beginning of Chapter 2. If the machine is turned off or DOS-32 disk pack unloaded, these instructions must be followed before DOS-32 can be used. Appendix F lists DOS system routines available to the user. The user must exercise care not to use any of these names for his subroutines or improper action may result. Appendix G describes how to use the DOS-32 overlay features. Appendix H lists error codes which may be encountered in FORTRAN compilation and execution. Appendix I gives a summary of system and library routines. Appendix J describes modifications made to the Honeywell supplied FORTRAN Library.

CONTENTS

Section		Page
1	THE DISK OPERATING SYSTEM - DOS-32	1-1
	1.1 INTRODUCTION	1-1
	1.2 DOS-32 FILE SYSTEM	1-3
	1.2.1 File Organization	1-3
	1.2.2 File Structure	1-4
	1.2.3 Multiple Disk Organization	1-5
	1.2.4 File System I/O	1-6
	1.3 DOS-32 CORE USAGE	1-7
	1.4 BREAK KEY INTERRUPT FEATURE	1-8
2	A SESSION AT THE CONSOLE	2-1
	2.1 INTRODUCTION	2-1
	2.2 INITIAL PROGRAM LOAD	2-1
	2.3 SYSTEM-USER INTERACTION	2-2
3	USER COMMANDS	3-1
	3.1 INTRODUCTION: COMMAND SYNTAX	3-1
	3.2 USER COMMANDS	3-3
	3.2.1 FTN: Compiling a FORTRAN Source Program	3-4
	3.2.2 MAC: Assembling a MAC Source Program	3-6
	3.2.3 EDIT: Editing a File	3-8
	3.2.4 MEDIA: File-Copy Utility	3-19
	3.2.5 BEDIT: Manipulating and Inspecting Files of Binary Programs	3-26
	3.2.6 LDR: Loading a File	3-30
	3.2.7 BUGGY: Debugging a Program On-Line	3-41
	3.2.8 CONCAT: Preparing Input for the Loader	3-46
	3.2.9 SUPDATE: Modifying a Source File	3-48
	3.3 I/O CONTROL COMMANDS	3-52
	3.3.1 INPUT: Opening an Input File	3-54
	3.3.2 BINARY: Opening an Object-Text File	3-54
	3.3.3 LISTING: Opening a Listing File	3-54
	3.3.4 OPEN: Opening Any File	3-55
	3.3.5 CLOSE: Closing a File or Unit	3-55
	3.3.6 COMINPUT: Switching Command Input from the Console to a File	3-56
	3.3.7 COMOUPUT: Switching Command Output from the Console to a File	3-58

CONTENTS (CONT.)

Section | Page

3.4 LOAD-MODULE MANAGEMENT COMMANDS............... 3-59

3.4.1 SAVE: Saving a Load Module............ 3-61
3.4.2 RESTORE: Reading a File into Core..... 3-62
3.4.3 START: Executing an In Core File...... 3-63
3.4.4 RESUME: Restoring and Starting a File................................ 3-63
3.4.5 PM: Displaying Program Status Words and Registers................... 3-64

3.5 USER DIRECTORY-HANDLING COMMANDS.............. 3-64

3.5.1 LOGIN: Getting Started................ 3-65
3.5.2 LOGOUT: Getting Off the System........ 3-65
3.5.3 STARTUP: Associating Disk Drives with DOS............................... 3-65
3.5.4 ATTACH: Gaining Access to a Directory. 3-66
3.5.5 LISTFILE: Listing the Files in a Directory............................. 3-67
3.5.6 DELETE: Deleting a File from a Directory............................. 3-67

3.6 UTILITY PROGRAMS............................. 3-67

3.6.1 LISTU: Listing the Source Files in a Directory....................... 3-67
3.6.2 CNAME: Changing the Name of a File.... 3-68
3.6.3 MOVEF: Moving a File from One Directory to Another.................. 3-68
3.6.4 MOVEFS: Moving Files from One Directory to Another.................. 3-69
3.6.5 COPYFS: Copying Files................. 3-69
3.6.6 CARDFS: Creating a File from a Card Deck............................. 3-70
3.6.7 PRINTFS: Listing Source Files on the Line Printer...................... 3-70
3.6.8 DMPSAV: Listing a Save File in Dump Format........................... 3-71
3.6.9 CPRSAV: Comparing Two Save Files in Dump Format........................ 3-71

4 SYSTEM ROUTINES AND UTILITY FUNCTIONS.............. 4-1

4.1 INTRODUCTION.................................. 4-1
4.2 FORTRAN INTEGERS.............................. 4-2
4.3 FILE-SYSTEM ROUTINES.......................... 4-3

4.3.1 OPEN: Opening a File.................. 4-4
4.3.2 CLOSE: Closing a File................. 4-6

CONTENTS (CONT.)

Section | Page

4.3.3 BRWFIL: Reading, Writing, or Manipulating a File.................... 4-6
4.3.4 DELETE: Deleting a File............... 4-9
4.3.5 REWIND: Repositioning a Unit at the Beginning of a File................ 4-9
4.3.6 ATTACH: Gaining Access to a Directory.............................. 4-10
4.3.7 COMANL: Performing Lexical Analysis of User Commands....................... 4-10
4.3.8 GETERR: Storing Error Information..... 4-12
4.3.9 PRTERR: Typing an Error Message....... 4-12

4.4 TELETYPE ROUTINES............................ 4-13

4.4.1 RDASR: Reading a Line from the Teletype.............................. 4-13
4.4.2 WRASR: Typing a Line on the Teletype.............................. 4-13
4.4.3 T1IN: Reading a Character from the Teletype.......................... 4-14
4.4.4 T1OU: Typing a Character on the Teletype.............................. 4-14
4.4.5 TNOU and TNOUA: Typing a Character String on the Teletype................ 4-14
4.4.6 TOHEX: Converting a Word to Hexadecimal and Typing It.............. 4-15

4.5 UTILITY FUNCTIONS............................ 4-15

4.5.1 AND: Forming Logical Product.......... 4-15
4.5.2 OR: Forming Logical Sum............... 4-16
4.5.3 XOR: Forming Exclusive OR............. 4-16
4.5.4 COMPL: Forming Complement............. 4-16
4.5.5 LH: Returning Left-Half of Word....... 4-17
4.5.6 RH: Returning Right-Half of Word...... 4-17
4.5.7 LHC: Returning Left-Half of Word, Indirect.............................. 4-17
4.5.8 RHC: Returning Right-Half of Word, Indirect.............................. 4-17
4.5.9 SLH: Storing Number in Left-Half of Word.............................. 4-17
4.5.10 SRH: Storing Number in Right-Half of Word.............................. 4-18
4.5.11 MAKWRD: Forming a Word from Two Halfwords............................. 4-18
4.5.12 LHWRD: Retrieving a Halfword.......... 4-18
4.5.13 SHWRD: Storing a Halfword............ 4-18
4.5.14 LCHAR: Retrieving a Character......... 4-18

CONTENTS (CONT.)

Section | Page

4.5.15 SCHAR: Storing a Character............ 4-19
4.5.16 PUTC: Inserting a Character in an Array................................ 4-19
4.5.17 LBIT: Retrieving a Bit................ 4-19
4.5.18 SBIT: Storing a Bit.................. 4-19
4.5.19 RT: Retrieving an Integer............. 4-20
4.5.20 SSWS: Testing a Sense Switch.......... 4-20
4.5.21 CHKSUM: Computing Checksum............ 4-20
4.5.22 COMEQV: Comparing Command-Names....... 4-20
4.5.23 NAMEQV: Comparing Names............... 4-21
4.5.24 DCNVRT/HCNVRT: Converting a Number from Internal Format to Decimal or Hexadecimal............................ 4-21
4.5.25 EXIT: Returning to the System......... 4-22
4.5.26 GTHEX: Searching for Hexadecimal Number in Character String............. 4-22
4.5.27 LHEX: Retrieving a Hexadecimal Digit................................. 4-22
4.5.28 GTREGS: Retrieving the Address of the Register Save Area................ 4-22
4.5.29 GTNAMS: Retrieving Directory and File from a Character String............... 4-23
4.5.30 LOC: Retrieving the Address of a Variable.............................. 4-23
4.5.31 LIST: Referencing an Absolute Core Address............................... 4-23
4.5.32 SRHC: Storing Number in Right-Half of Word, Indirect.................... 4-24
4.5.33 SLHC: Storing Number in Left-Half of Word, Indirect.................... 4-24

4.6 PERIPHERAL DEVICE ROUTINES.................. 4-24

4.6.1 DOCARD: Reading a Card............... 4-25
4.6.2 DOPRIN: Printing a Line.............. 4-25
4.6.3 MT7IOC: Controlling Magnetic Tape..... 4-25
4.6.4 P1IN: Reading a Character from Paper Tape........................... 4-28
4.6.5 P1OU: Punching a Character............ 4-29
4.6.6 PNOU: Punching a Character String..... 4-29
4.6.7 P1INB: Reading a Binary Character from Paper Tape....................... 4-29
4.6.8 P1OUB: Punching a Binary Character.... 4-29
4.6.9 PNOUB: Punching a Binary String....... 4-30

4.7 USING FORTRAN INPUT-OUTPUT STATEMENTS UNDER DOS.................................... 4-30

4.8 OVERLAY-MANAGEMENT ROUTINES.................. 4-32

CONTENTS (CONT.)

Section — Page

4.8.1 RESTOR: Bringing a Load Module into Core 4-32

4.8.2 SAVE: Storing a Load Module on the Disk 4-32

Appendix A: Glossary for DOS-32 .... A-1

B: Summary of User Commands B-1

C: Command-Name Abbreviations C-1

D: Programming Hints and Warnings D-1

E: Initialization Instructions E-1

F: DOS System Routines F-1

G: Overlay Structures G-1

H: Fortran Error Messages H-1

I: Summary of System and Library Routines I-1

J: Fortran Library J-1

ACKNOWLEDGMENT

In preparing this document, I have acted more as an editor than a writer. The following people wrote the bulk of the document:

Ms. Ilene Lang

Ms. Marsha Siegal

Mr. John Thron

Mr. George Touchstone

Mr. Robert Wadsworth

Mr. Carey Wyman

The system described by this document is a creation of the people named above and the following:

Mr. Barton Bruce

Mr. John Keen

Dr. John William Poduska

Mr. David Udin

Others, too numerous to mention, were also involved.

CHAPTER 1
THE DISK OPERATING SYSTEM - DOS-32

1.1 INTRODUCTION

DOS-32 was developed for the Honeywell CCD Series-32 computers, the 32-bit word H632 and H832. It is an interactive operating system which provides an environment for the development of system and applications programs. Users of DOS-32 are able to create and manipulate a data base with ease, to incorporate interactive capabilities in FORTRAN and MAC programs, and to control use and maintenance of system resources from the console.

DOS-32 consists of a file system, a set of system programs and utility routines, a set of hardware-level I/O routines, and an interactive command language. The file system simplifies the creation, deletion, and updating of source, object, load-module, and data files, and provides great flexibility in accessing the data base. The system programs and utility routines include a FORTRAN compiler, a MACRO assembler, a loader, an interactive editor, an on-line debugging program, and a generalized routine that copies data from one storage medium to another. They can be invoked and controlled by simple commands issued from the console and are fully integrated with the file system, which they use for all input and output. The input/output routines can be called from user programs to perform I/O operations at the record or character level. They include teletype-control routines; user programs can therefore be interactive. The DOS-32 interactive command language allows you to perform maintenance operations on files and to control the execution of all system programs and utility routines.

DOS-32 provides two levels of user-system interaction. The first is the supervisory level, in which you issue instructions (user commands) to the system by typing them on the console typewriter. These user commands invoke various capabilities of the system, such as system programs and maintenance facilities. Thus, for example, you can type the command

```
FTN
```

and thereby invoke the FORTRAN compiler to compile a source program; or you can type the command

DELETE *file*

and thereby erase *file* by releasing the storage it occupies and removing its entry from your directory. As a DOS user, you can create programs, modify them, compile, load, execute, and store them, simply by typing the appropriate user commands at the console.

The second level of user-system interaction is the runtime level, in which a program is in execution. This program invokes system I/O routines, file-manipulation routines, and utilities by calling the appropriate routine, e.g., by specifying it in a FORTRAN CALL or function statement. Thus, for example, a program can contain the statement

```
CALL TNOUA ('FILE NAME = ',12)
```

and the DOS routine TNOUA will type out the specified 12-character literal on the teletype.

Central to DOS-32 is its file system. In both modes of user-system interaction, you deal with the file system. The file system provides a mechanism for storing organized collections of data under a name which you specify, and for retrieving that set of data at some later time. A file can contain a source program, an object program, a load module, or data. Physically, a file consists of records, which the system formats and keeps track of for you. In addition, the file system maintains various directories, in which files are listed and through which they are accessed. The file system provides a set of file-manipulation routines which, among other things, allow you to enter new files in a directory, and list the contents of a directory.

1.2 DOS-32 FILE SYSTEM

1.2.1 File Organization

A file is simply an ordered linear array of words known by a name. A file may contain a source program, an object program, a load module, data records or other information.

The files on a disk are recorded as chained strings of fixed length records. The first two words of each record are reserved for DOS. The left half of the first word points to the predecessor record (zero, if none), the right half of the first word points to the successor record (zero, if none) and the left half of the second word is a count of the number of bits of data contained in the record. The rest of the record contains data of the file. When the user reads a file, only the data portion of the file is given to him. The user is not aware of the record structure of the file; he knows the file as a linear array of words.

The collection of files belonging to a user is organized into a User-File-Directory (UFD). Each UFD is a file and contains a block of information for each file which includes the file name and starting record address of the file.

The collection of all UFD's on one disk is organized into a file directory called the Master-File-Directory (MFD). The MFD contains a block of data for each UFD which includes its name and starting record address. In addition, the MFD contains a special file DSKRAT which is a table of disk allocation information for the disk pack. The file system must select free records when a file is to be written or extended, and the records of a deleted file must be returned for later re-use. Records are selected and returned using this table, which contains one bit for each record of the disk. The disk contains 8120 records, so the DSKRAT table is about 508 words in size.

1.2.2 File Structure

DOS provides for reading and writing files in two modes - word mode and line mode. In word mode, the user specifies that N words are to be read from the current word pointer of a file into an array in core. Repeated calls to the DOS routine BRWFIL for reading N words will cause N word blocks of the file to be read.

To write in line mode, the user specifies that an array or line of N words of characters, 4 characters to a word is to be written in a compressed mode. The line must not contain a new line (linefeed) character. Whenever a series of 3 or more spaces occur, they are replaced by the character with the octal representation 21 (the ASC11 character DC1), followed by a count of spaces. Trailing blanks, if any, are eliminated and a line feed character is added. The line is then written. Because of the compression, the last word of the compressed line may contain less than four characters. The BRWFIL routine writes only the number of characters to be written even if writing must stop in the middle of a word.

To read a file in line mode, the user specifies that the next line of the file is to be placed in an array in core of size N words or 4xN characters. The next line of the file is read and uncompressed into the array. The length N of the array to be filled must be large enough to receive the line as written. A short line will be padded with blank characters to fill the array. Once a file has been written in line mode, it must be read in line mode. Similarly, once a file has been written in word mode, it must be read in word mode.

1.2.3 Multiple Disk Organization

DOS-32 can handle either one or two disk drives. Each disk pack contains its own Master-File-Directory and DSKRAT. All files in a user file directory reside on that disk and all user file directories in the MFD reside on that disk.

If one drive is used, it must be drive 0, as the disk hardware bootstrap always attempts to load the operating system from that drive. The STARTUP command tells the system how many drives are in use and specifies a default order of searching MFDs in attempting to ATTACH to a directory. For example, STARTUP 1 0 tells the system 2 drives are active, and that the command ATTACH UTIL will attach to UTIL on drive 1. If there were no UTIL on drive 1, it would look for UTIL on drive 0. The search order may be overridden by supplying the drive number in the ATTACH command. Thus ATTACH UTIL 0 would look for UTIL on drive 0. If UTIL is not on that drive, no other drives are searched and an error message is given.

External commands such as EDIT, FTN and MAC which are core image files are resumed from the COMDIR directory on drive 0. Thus only one copy of the system commands must be stored on a set of disk packs, on the master pack.

If only the master disk pack is in use on drive 0, the appropriate STARTUP command (STARTUP 0) need not be given when the user loads the system.

A user may have more packs than drives, in which case he may wish to unmount one or more packs and mount others. All packs may be unmounted. The sequence of commands is LOGOUT followed by the appropriate STARTUP command for the new configuration of packs. The LOGOUT command must be given before the disk pack or packs are unmounted.

Files may be active on two disks at once, allowing files to be moved from one disk to another with ease. COPYFS, a user command program has been written for this purpose.

1.2.4 File System I/O

Although the file system can contain an indefinite number of files, only seven may be active at any one time. A file becomes active when you connect it to a UNIT represented by the integer 1-7 which functions as a port to the system. One file at a time may be assigned to each unit, at which time the mode of processing is specified (read, write, read and write). Files may be active on two disks at once. The transfer of data to and from a file through a unit is accomplished by the BRWFIL routine.

Many DOS commands refer implicitly to specific units. The command INPUT opens unit 1 for the input file, BINARY opens unit 3 for output and LISTING opens a listing file on unit 2.

Users may call on the file system to manipulate files as described in section 4.3.

1.3 DOS-32 CORE USAGE

DOS is a core resident operating system. DOS occupies locations 200 to 4EFE hex. Locations below 200 are reserved for interrupt pointers and CP and IOP status words. When a user is loading programs, the loader must be resident. The loader occupies locations 4F00 to 63FE hex. DOS disk I/O buffers occupy locations F280 to FFFE hex, leaving a user 6400 to F27E hex for his programs.

The DOS buffers are only allocated as needed, from the top of memory downward. If the user's program does not use seven units at once, his program may occupy more core. DOS requires one system buffer plus one buffer for each open file. Each buffer occupies 1B0 hex locations.

A user may specify a load address of 4F00 hex. The loader will start loading at 6400 hex but generates relocated addresses as if the load was made starting at 4F00. When control is returned to DOS, the entire load is "pulled down" over the loader. Although no larger programs can be loaded using this feature, the extra space made available at the top of memory may be used for DOS buffers or free storage by the user program.

1.4 BREAK KEY INTERRUPT FEATURE

Following initial program load of DOS-32, the user can control operation of the H632 System entirely from the console teleprinter. When the <u>break</u> key on the teleprinter is depressed, program execution is immediately interrupted, and one of the following operations must be selected by the user:

1) Resume program execution
2) Set a specified pseudo sense switch (and resume program execution)
3) Reset a specified pseudo sense switch (and resume program execution)
4) Return control to the DOS supervisor
5) Reinitialize the system and return control to the DOS supervisor

(Register and storage access operations can be performed from the console teleprinter using the command BUGGY.) Any unsolicited type-in will also cause program execution to be interrupted. The console interrupt service routine invites a system control command by typing the sequence carriage return, line feed, question mark, space. At this point, any type-in other than a defined system control command causes the above-described invitation to be repeated. System control commands are described in the remainder of this section

<u>[Space]: Resuming Program Execution</u>

Depressing the space bar causes a type-out of period, space, CR, LF, followed by resumption of program execution at the point of interruption.

S: Setting a Specified Pseudo Sense Switch

The typing of the SET command name is automatically completed by the output of ET and a space. At this point, any type-in other than a defined pseudo sense switch identifier causes the command invitation type-out to be repeated. If the user types one of the letters A through O, the corresponding pseudo sense switch is assigned the value 1 (.TRUE.); then the actions specified for space (see above) are performed. To check a pseudo sense switch from a user program, the user calls the logical function SSWS(N) where N is 1-15 corresponding to pseudo sense switches A-O. SSWS returns .TRUE. if the switch was set, .FALSE. otherwise.

R: Resetting a Specified Pseudo Sense Switch

The typing of the RESET command name is automatically completed by the output of ESET and a space. Command execution is thereafter exactly as for S, except that the identified pseudo sense switch is assigned the value 0 (.FALSE.).

Q: Returning Control to the DOS Control Program

The typing of the QUIT command is automatically completed by the output of UIT followed by a space and the sequence period, space, CR, LF. Control is then transferred to EXIT, resulting in the familiar 'OK,' solicitation of a new DOS command.

A: Reinitializing the System

The typing of the ABORT command is automatically completed by the output of BORT. A programmed system initialize is performed. Then all open files are closed. Finally, the sequence space, &, space, Q is typed and control is passed to the QUIT command processor (see above).

CHAPTER 2
A SESSION AT THE CONSOLE

2.1 INTRODUCTION

This chapter illustrates a typical set of tasks you will perform at the console. During this sample console session, you will open files, compile subroutines, utilize list options, transfer data from cards onto disk and into core, and debug. It should be noted that this chapter presents representative procedures; procedures performed by the individual user may vary considerably from those illustrated in the session below. The intention of this chapter is to show a large collection of typical tasks and thus illustrate for many users the varied capabilities of DOS-32.

2.2 INITIAL PROGRAM LOAD

Before logging into the system and beginning work, the user should check that the DOS USER disk is running on drive 0 and DOS is loaded. If you are following another user, the last thing typed should be LOGOUT followed by OK. If the system is not running, the user must load DOS by means of the Initial Program Load or IPL procedure as explained below and in Appendix E.

1) Set the four Device Select switches on the System Control Panel, from left to right, to ON, OFF, ON, ON. (Down is ON.)
2) Using the key-operated Power switch, ensure that power is ON and the switch is in the UNLOCK position.
3) Place the DOS user disk on disk drive #0 of the disk control unit #0, close the protective cover, and depress

the Start switch on the disk drive #0 switch panel. Then, depress the Permit switch on this switch panel.

4) After the Ready light on the disk drive illuminates, depress the System switch on the SCP, wait for a half-second, and then depress the Start switch on the SCP. The teletype should respond by typing "DOS-32 REV G" followed by "DATE=".

5) User types date in any format less than eight characters. DOS responds "TIME=".

6) User types time in any format less than eight characters. DOS responds by typing "OK,".

2.3 SYSTEM-USER INTERACTION

The following is a typical console session

```
user:  LOGIN WYMAN
response:  OK
```

Before we can begin work, we must LOGIN or ATTACH to one of the user file directories. LOGIN is one of many internal commands of DOS. WYMAN is the name of one of the user file directories of DOS. The command LOGIN WYMAN searches the master file directory MFD for WYMAN, and if found establishes that directory as the current user file directory. When attached to the user file directory, we have access to all files in that directory and may generate new files in it if we wish.

```
user:  load program in card reader
user:  CARDFS
response:  >
user:  PROG
```

```
response:  >
user:  SUBR
response:  OK
```

The user has prepared a FORTRAN main program and one subroutine called by the program on cards of EBCDIC code. Each program is followed by a $EOF card with an extra $EOF card at the end of the deck.

CARDFS is the name of an external command. When a user types a command to DOS, DOS looks the word up in a list of internal command words. If the name is not in the list, DOS looks the name up in a special external command directory COMDIR. If the name is found, DOS reads in a core image file and starts executing the program with initial PSW1, PSW2 and 16 registers as specified in a special part of the file. An external command, in other words, is a command which causes a core image file to be read into core and begin execution.

CARDFS is loaded into core memory from COMDIR, and begins execution by typing >. The user gives the name of a file to be assigned to the first program. CARDFS creates the file and writes the card images of the first program into that file, in this case PROG. CARDFS then types > and waits for the name of another file. The user typed SUBR and CARDFS wrote the second program onto that file, then returned to DOS upon detecting two $EOF cards in a row. No $EOF lines are placed in the file. All file names must be 1-8 alphanumeric characters beginning with a letter.

```
user:  INPUT PROG
response:  OK
```

We are preparing to compile PROG with FORTRAN. The INPUT command connects the file that is its argument to unit 1 for reading. In order to read or write files, they must first be connected to a unit, which functions as a port to the system. CARDFS connected files to units under program control, but FORTRAN expects the file to be compiled to be connected to unit 1 by the user.

```
user:  BINARY  B:PROG
response:  OK
```

The BINARY command connects the file that is its argument to unit 3 for writing. If the file did not exist, it will be created by this command. FORTRAN expects unit 3 to be open for writing and writes binary compiled output through unit 3.

```
user:  FTN ALIST
response:  listing on line printer
           0 errors
           OK
```

FTN is the external command which calls the FORTRAN compiler to compile a program. ALIST is a FORTRAN option which causes a list of variables and their relative locations within the program to be printed. The program PROG is compiled with listing on the line printer and binary output to file B:PROG. After compilation is complete, FTN automatically disconnects all units from files and returns to DOS, which types OK.

```
user:  I SUBR, B B:SUBR, FTN
response:  0 errors
           OK
```

The internal commands may be abbreviated. INPUT is abbreviated I, and BINARY B. Each internal command may be abbreviated by truncating characters from the right in such a way that the abbreviation is unique. See appendix C for a complete list of internal commands and their abbreviations.

Multiple commands may be stacked on the same line, each separated by a comma.

The group of commands given causes SUBR to be compiled in a similar way that PROG was compiled.

```
user:  EDXX? EDX"IT
response:  INPUT
```

The editor is invoked by the external command EDIT. The "?" character kills all input on the line up to the "?". The " character erases the last character typed before it. Successive use of " causes successive preceding characters to be erased. The editor has just typed INPUT indicating the editor is in high speed input mode. In this mode, typed lines are entered directly into the text buffer.

```
user:  B:PROG B:SUBR
user:  carriage return
response:  EDIT
           >
```

Two carriage returns in succession cause the editor to switch modes. We have switched to EDIT mode, in which typed lines are interpreted as commands to the editor. When in the EDIT mode, a ">" is typed at the beginning of the line, indicating the editor is ready to accept a command.

```
user:  FILE N:PROG
response:  OK
```

FILE is an editor command which writes a file that contains the text buffer as data and with the name that is its argument. The FILE command returns control to DOS at its completion. DOS types OK

```
user:  CONCAT
response:  OUTPUT FILE=
user:  C:PROG
response:  NAME FILE=
user:  N:PROG
response:  NAME FILE=
user:  Q
response:  OK
```

It is generally convenient to maintain subroutines of a program as separate files. Thus instead of always compiling all subroutines in a program, one need compile only those which are being changed. To facilitate loading, CONCAT is used to combine all subroutines of a program into a single output file which is acceptable to the loader.

CONCAT requests the output file which the user responded by giving C:PROG. CONCAT then requests a name file which is a file that contains the names of all files to be combined. The user typed N:PROG, the name of the file containing the two names B:PROG and B:SUBR. CONCAT asks the user for another name file to add more routines to the output file. The user typed Q for QUIT to indicate there were no other files to be concatenated and to cause a return to DOS

```
user:  LDR
response:  >
```

LDR invokes the relocating loader. LDR types > indicating it is ready to accept requests to load programs.

```
user:  *,PRI, C:PROG, LIB, REF, MAP
response:  file is loaded, load map is typed and printed on the
           line printer
           FUL= 7500
           >
```

The user typed line indicates the following:

*	- begin loading at the first available location
PRI,C:PROG	- load file C:PROG as the primary file
LIB	- search the file LIBRARY in user file directory LIBDIR to satisfy unresolved references
REF	- continue processing of the loader, even if unresolved references remain
MAP	- print out all entry points and unresolved references

LIBRARY contains all the mathematical, conversion and I/O routines required by FORTRAN. In addition, utility functions and routines described in section 4.5, and peripheral device routines described in section 4.6 are in LIBRARY. The loader automatically resolves references to resident DOS routines described in sections 4.3 and 4.4.

```
user:  SAVE *PROG 6400 7500 6410
response:  OK
```

SAVE generates a core image file named *PROG containing locations 6400 to 7500 as data. SAVE also generates a vector of machine initial conditions for restoration when the file is resumed, putting this information at the beginning of the file. These conditions are given beginning as the fourth parameter of the

SAVE command. They are in order PSW1, PSW2, and R0 through R15. In this case only PSW1 was specified.

The user's save file should start at 6400 unless he has specified a different load address to the loader. The end address should be the address indicated following FUL= on the load map. FUL stands for first unused address. The initial PSW1 or entry point should be specified as the address following MAIN in the load map.

```
user:  CARDFS
response:  >
user:  DATA
response:  OK
```

The user uses CARDFS to read a data card deck onto disk as file DATA.

```
user:  INPUT DATA
response:  OK
user:  RESUME *PROG
response:  output to line printer
           OK
```

The user connects unit 1 to file DATA by the INPUT command. RESUME is a DOS internal command that reads a core image file into core, sets the machine registers from a part of the file, and begins execution. The program reads input from unit 1 writing output on the line printer. When the program finishes, control returns to DOS and OK is typed.

```
user:  CLOSE ALL
response:  OK
```

CLOSE ALL will disconnect all units from all files.

To allow the user to try the procedure described in this section, the programs PROG and SUBR are given below

```
C      FILE PROG
C
       DO 10 I=1,10
       READ(1,20) J,K
20     FORMAT (2I6)
       CALL SUB (J,K)
10     CONTINUE
       CALL EXIT
       END
$EOF
C      FILE SUBR
C
       SUBROUTINE SUB (J,K)
C
       L = J + K
       WRITE (101,20) L
20     FORMAT (1X,I6)
       RETURN
       END
$EOF
$EOF
```

In the above program, the statement READ(1,20) J,K will read from DOS file unit 1. The FORTRAN READ device number corresponds to a DOS file unit number for the range 1 to 7. In the WRITE statement, device 101 will output to the line printer. The subroutine EXIT will return control to DOS which types "OK". The data cards should consist of 10 cards followed by $EOF each containing two integers right justified to columns 6 and 12.

An alternate way of using the system is to generate program and data files at the teletype using the editor. If the program is short and consists of only a few subroutines, it may be quicker to bypass use of CONCAT. The following is an example of this

```
user:  EDIT
response:  INPUT
```

The user invokes the EDITOR as in the first example

```
user:      DO  10  I = 1,10
user:      READ (100,20) J,K
user:  21 \ FORMAT
user:      CALL SUB(J,K)
user:  10 \ CONTIN
user:      CALL EXIT
user:      END
user:  $EOF
user:      SUBROUTINE SUB(J,K)
user:  C
user:      L = J+K
user:      WRITE (100,20) L
user:  20 \ FORMAT (I6)
user:      RET?    RET"TURN
user       END
user:  $EOF
user:  $EOF
user:  carriage return
response:  EDIT
           >
```

The backslash character "\" is the logical tab convention for EDIT. We have entered a program and subroutine using the INPUT mode of the editor. When finished, two carriage returns in a row were typed to enter the EDIT mode, in which typed lines are interpreted as commands to the EDITOR. The ">" symbol is typed whenever the editor is ready to accept commands in the EDIT mode. After a command is given, the editor takes time to execute the command. The user should not type another command until the ">" is given. This symbol also makes it easy to remember if one is in INPUT or EDIT mode, as ">" is not typed in INPUT mode.

```
user:  TOP
response:  >
```

TOP is a command which moves a pointer which points to lines of text in the text buffer. This pointer is moved to point to a "null" line at the beginning of the text buffer.

```
user:  FIND 31
response:  BOTTOM
           >
```

The FIND command moves the pointer forward from the line following the current line to the first line beginning with the string "31". The pointer now points to a "null" line after the last line of text in the buffer. What I meant to do was type FIND 21 but mistyped it.

```
user:  TOP
response:  >
user:  FIND 21, PRINT
response:  21  FORMAT
           >
```

This sequence of commands means go back to the top of the buffer, look for a line beginning with "21", then print the line.

The second user input shows stacking of commands. Two commands have been typed on the same line separated by a comma. The commands are not executed until the entire line has been typed. The commands are then executed left to right

```
user:  RETYPE 20\ FORMAT(2I6)
response:   >
user:  PRINT
response:  20    FORMAT(2I6)
           >
```

RETYPE replaces the current line by the string that is the argument of the retype command.

```
user:  LOCATE  CONTIN, PRINT
response:  10  CONTIN
           >
```

LOCATE repositions the current line pointer to the next line in the text buffer containing an occurrence of the string that is its argument

```
user:  CHANGE  /CONTIN/CONTINUE/,PRINT
response:  10  CONTINUE
           >
```

The CHANGE command is used to replace one string of characters in the current line by another. "/" is used in this case as the delimiter of the strings, but almost any character will do.

```
user:  FILE  TEST
response:  OK
```

As before, the FILE command writes the text buffer onto the file which is its argument, then returns control to DOS.

This program is different from the first example, as device 100 is used for reading and writing. Device 100 is the teletype.

```
user:  INPUT  TEST, BINARY  B:TEST, FTN
response:  0 errors
           OK
```

Compile the program TEST with binary output B:TEST with output listing on the line printer. The compiler will compile "stacked" subroutines so long as each is separated by a $EOF card with two $EOF cards following the last subroutine. Output for both the main program and the subroutine in this example is written onto B:TEST.

```
user:  LDR
response:  >
user:  *,PRI,B:TEST, LIB,REF,MAP
response:  >  (file B:TEST is loaded, map is printed)
user:  carriage return
```

Binary file B:TEST is loaded as C:PROG was in the first example. The user can speed up output of the load map by telling the loader to print it only on the line printer by setting pseudo sense switch A. This is done by hitting the BREAK key, and typing S, then A. Pseudo sense switch A may be reset by hitting the BREAK key and typing R followed by A. The sense switch should be set before the LDR command is given. The BREAK key feature has other purposes described in section 1.4.

```
user:  SAVE  *TEST  6400  7500  6410
response:  OK
```

Generate the core image file *TEST containing locations 6400 to 7500 with entry point 6410. As before, the first location saved should be 6400, the last location should be that following FUL= of the load map, and the entry point should be that location opposite MAIN in the load map.

```
user:  RESUME  *TEST
response:  none
```

The program is waiting for teletype input.

```
user:  00001  00005
response:  6
user:      00100  00300
response:  400
user:  BREAK key
response:  ?
user:  Q
response:  OK
```

The user wishes to abort his program and return to DOS command level. He hits the BREAK key, which causes a program interrupt and a special DOS routine to be entered. This routine accepts a limited number of special commands. The Q command is given, which causes control to return to DOS command level. See section 1.4 for a full description of the BREAK feature.

```
user:  LOGOUT
response:  OK
```

The user gives the LOGOUT command at the end of his session. This disconnects all units from all files, updates the disk and detaches DOS from your directory. This prevents the next user from accidently using your directory.

CHAPTER 3
USER COMMANDS

3.1 INTRODUCTION: COMMAND SYNTAX

A *user command* is what you type at the console to instruct DOS to perform some action; it consists of a key word, followed by a string of arguments, followed by a terminator. The number and nature of the arguments depend on the command itself. The *terminator* is either a comma or a carriage return: use a comma if you want to type another command on the same line; use a carriage return if you want to type the next command on a new line.

A *key word* is an alphanumeric string which has special meaning to the system. It begins with an alphabetic character and is followed by a delimiter (blank, comma, left parenthesis, right parenthesis, or carriage return); see Appendix B. An *argument* is either a name or a parameter. A *name* is an alphanumeric string which begins with an alphabetic character and is not a system key word. A *parameter* is a number which is represented in either hexadecimal or octal notation. Hexadecimal numbers must begin with a digit 0-9 (for AA type OAA); octal numbers must begin with an apostrophe ('). Although arguments can technically be any length, only the first eight characters of a name or the rightmost 32 bits of a parameter are used. Arguments must be separated from each other by at least one blank.

Commands can be typed in a fairly free format. Two or more commands can be typed on a single line by separating them with commas. Commands can be continued from one line to the next by typing a semicolon (;). A carriage return which is not preceded by any semicolon signals the end of a line. When you type a carriage return, DOS analyzes and executes the commands that you have typed on the line, one at a time. If there are no syntax errors in the line, and if each command is executed properly, DOS types OK. If errors are discovered, DOS types an error message and the code ER,; it then returns the carriage to the beginning of the next line (having processed commands up to the error and ignoring any other commands in the current line) and awaits a new command. You can erase the most recently typed characters one at a time by typing one quotation mark (") for each character you wish to delete. You can delete an entire line by typing a question mark (?).

The DOS user commands are presented in detail in the sections of this chapter. In the models, the following conventions are used. Words in CAPITAL LETTERS are key words. Items in *lower-case italics* are arguments. Parentheses are required where shown. Elements in square brackets [] are optional; elements stacked within {} are alternatives, of which one must be chosen; and an ellipsis ... indicates that the preceding element may be repeated. A carriage return is specified as ↵ .

There are two basic types of commands, internal commands and external commands. Internal commands are commands executed by the core resident DOS. After a command is typed, DOS looks it up in its list of command names. If found, the appropriate action is taken and control is returned to DOS command level. If the name is not found, DOS looks the name up in a special external command directory COMDIR. If the name is found, DOS reads in the core image file and starts executing the program with initial program status words and registers as specified in the beginning of the file. An external command, in other words, is a command which causes a core image file to be read into core and begin execution. System programs and utility commands are external commands; all others are internal. Each internal command may be abbreviated by truncating characters from the right in such a way that the abbreviation is unique. See Appendix C for a complete list of internal commands and their abbreviations.

3.2 SYSTEM PROGRAMS

DOS-32 currently includes nine system programs: a FORTRAN compiler (FTN), a MACRO assembler (MAC), a Loader (LDR), a program for building loader input (CONCAT), an interactive editor (EDIT), an object-program editor (BEDIT), a media converter (MEDIA), an on-line debug (BUGGY), and a batch-mode update program (SUPDATE). Although DOS-32 has ultimate control over the machine, it delegates control to system programs when they are invoked. Communication then is between you and the system program. Thus, for example, you can invoke the editor by typing the command

```
EDIT
```

The editor then takes over, and you issue editor commands which the editor interprets and executes. When you are finished using the editor, you can get back to DOS by typing

```
QUIT
```

This is a signal for the editor to relinquish control, which passes back to DOS. Subsequently you will type user commands and will be interacting with the system. If your next command is

```
MEDIA
```

the media conversion program will take control, and you will issue MEDIA commands, etc.

3.2.1 FTN: Compiling a FORTRAN Source Program

Model:	FTN [*option*]...	
Examples:	B B:SIGMA, FTN	*Input on cards, listing on printer, binary on disk (B:SIGMA)*
	FTN NOBJ	*Input on cards, listing on printer, no binary*
	I S:DELTA	*Input from disk (S:DELTA)*
	B B:DELTA	*('S:' and 'B:' have no special signi-*
	FTN ALIST OLIST	*ficances; they are chosen by user to help him identify different types of files)*

FTN invokes the FORTRAN compiler to compile a FORTRAN source program. Input may be on cards (default) or on disk, in which case the INPUT command (3.3.1) must be used to identify the file to the compiler. The listing may be sent to the printer (default) or to the disk via the LISTING command (3.3.3). The object program will be written on the disk into the file named in the BINARY command (3.3.2), unless the NOBJ option has been specified. When compilation is complete, the compiler will close all files and return to DOS, which will type OK,.

Several programs may be compiled by one invocation of the compiler provided they are separated by $EOF records. Object programs will be written successively into the binary file without separations. If input is on cards, two successive $EOF cards represent end-of-file. After each of the programs has been compiled, FTN will type a message telling how many errors were found.

When a group of statements, such as a set of COMMON statements, appears in several programs, this group may be stored in a separate file and replaced by a single statement in each program which reads:

INSERT *(directory password) filename*

where *directory* and *password* are optional. On encountering such a statement, FTN will process the statements in *filename* as if they had actually appeared in place of the INSERT statement.

Any combination of the following options (separated by blanks) may be specified in the command line:

Option	*Effect*
SLIST	List source program (default).
NSLIST	Do not list source program.
OLIST	List object program by printing generated MAC statements after each FORTRAN statement.
NOLIST	Do not list object program (default).
ELIST	List external references (default).
NELIST	Do not list external references.
ALIST	List storage assignments.
NALIST	Do not list storage assignments (default).
OBJ	Generate object program (default).
NOBJ	Do not generate object program.
ST	Symbol table (not used by DOS-32)
NST	No symbol table (default).
DP	Double-precision mode for all REAL computation and data allocation.
NDP	Normal mode (default).
X	Compile X lines.
NX	Do not compile X lines (default).

The use of X and NX options may be unclear to some users. An X may be placed in column 1 of a FORTRAN statement to indicate the temporary nature of this statement. If the X compiler option is specified, lines containing an X in column 1 will be compiled;

otherwise, these lines will be treated as comments. This facility is very useful for debugging purposes.

Any line with 'C1 ' in positions 1-4 is treated in a special way by FTN. It is not printed, and the line which follows it is printed on a new page. Errors detected during compilation are printed out in code form after compilation is complete. Each follows the statement in which it is encountered (these codes are interpreted in Appendix H).

3.2.2 MAC: Assembling a MAC Source Program

Model:	MAC [*option*]...	
Examples:	B B:SIGMA, MAC	*See FTN for comments on these examples*
	MAC NOBJ	
	I S:DELTA, B B:DELTA, MAC XF SET	

MAC invokes the MACRO assembler to assemble a MAC source program. Input may be on cards (default) or on disk, in which case the INPUT command (3.3.1) must be used to identify the file to the assembler. The listing may be sent to the printer (default) or to the disk via the LISTING command (3.3.3). The object program will be written on the disk into the file named in the BINARY command (3.3.2), unless the NOBJ has been specified. When assembly is complete, the assembler will close all files and return to DOS, which will type OK,.

Several programs may be assembled by one invocation of the assembler, provided they are entered on cards and separated by $EOF records. (If input is on disk, only one subroutine can be assembled at once.) Object programs will be written successively into the binary file, without separators (this is not recommended).

Two successive $EOF cards represent end-of-file. After each of the programs has been assembled, MAC will type a message telling how many errors were found.

When a group of statements, such as a set of COMMON statements, appears in several programs, this group may be stored in a separate file and replaced by a single statement which reads:

INSERT *(directory password) filename*

where *directory* and *password* are optional. On encountering such a statement, MAC will process the statements in *filename* as if they had actually appeared in place of the INSERT statement.

Any combination of the following options (separated by blanks) may be specified in the command line:

Option	*Effect*
LO	List source program (default).
NLO	Do not list source program.
OBJ	Generate object text (default).
NOBJ	Do not generate object test.
SC	Symbol concordance (default).
NSC	No symbol concordance.
ST	Symbol table (not used by DOS-32).
NST	No symbol table (default).
XF	Output X and K error flags.
NXF	Do not output X and K error flags (default).
FPL	Pass-1 listing.
NFPL	No pass-1 listing (default).
SET	Enter SET source lines in concordance.
NSET	Do not enter SET source lines in concordance (default).
TXS	Right-fill text with $20 (default).
TXN	Right-fill text with $00

Assembly errors are identified by one-character codes printed to the left of the relevant lines in the assembly listing. These codes are explained in the *MAC-32 Assembler Manual*. A special class of system-related errors are reported after assembly by a message on the console in the form:

CO$ MAC$ *code*

where *code* represents one of the following:

Code	*Meaning*
T01	Bad control card
T02	Illegal I/O request
T03	Symbol table too large
T04	Bad load of MAC
T05	No end-of-group
T06	Bad read

3.2.3 EDIT: Editing a File

Model:	EDIT [*file*]
Examples:	EDIT
	EDIT BIN1

EDIT invokes the DOS-32 interactive text editor, which allows you to edit files according to context and to create, edit, and store new files.

The editor provides two modes of operation, input and edit. If you are in input mode, lines which you enter at the teletype are treated as lines of a new file which you are creating. If you are in edit mode, lines which you enter at the teletype are treated as editor commands which are to be executed on an existing file. At any point, you can switch from one mode to the other by typing two consecutive carriage returns or line feeds. Thus, if you want to add several successive lines, you can use input mode rather than the INSERT command.

The editor processes a file, which consists of lines - strings of characters delimited by new-line characters (carriage returns or line feeds). You have access to a file one line at a time and can specify commands to be executed on that line. The current line is indicated by a pointer, which can be positioned at any line in the file. Various commands can be used to move the pointer up and down. The pointer is not affected by switching between edit and input modes.

The editor is invoked when you type the command:

```
     EDIT [file]
```

where the optional *file* designates a file in the current directory. If *file* is specified, you will enter the editor in edit mode. The editor will respond by typing

```
     EDIT
```

and positioning the pointer at the first line of the named file (the first line is always a pseudo-line: '.NULL.'). You then issue commands (see below) to edit that file. When the end of that file is reached, the editor will type

```
     BOTTOM
```

and you can continue in any of a number of ways, depending on the commands which you subsequently issued.

If you invoke the editor and do not specify a *file*, you will enter the editor in input mode. The editor will respond by typing

```
     INPUT
```

You then type lines which will be entered in the editor buffer as a file. You can eventually enter this file in the current directory. When you are finishing entering lines, enter two new-line characters. This will signal the end of the file, in response to which the editor will put you in edit mode. The editor will then type

```
     EDIT
```

and the pointer will point at the last line of the file. You can then process the file with any of the commands described below.

Editor commands are specified in command lines, according to the following format:

command [,*command*] ...

where each *command* is one of the editor commands described below. When you signal the end of a command line by typing a new-line character, the editor will begin to execute the commands on the line.

The ">" character is typed whenever the editor is ready to accept commands in the EDIT mode. After a command is given, the editor takes time to execute the command. The user should not type another command until the ">" is given. This symbol also makes it easy to remember if one is in INPUT or EDIT mode, as ">" is typed only when in EDIT mode.

To facilitate ease of input, the editor makes use of tab stops. These stops are originally set at 6 and 12 characters from the left, but by using the TABSET command up to 8 tabs can be set anywhere along the line. To use the tab feature, the user types a backslash (\). This character is interpreted as meaning fill in this line with blanks until the next tab stop. Hence, if one wishes to input the following FORTRAN code: (periods indicate spaces in line).

```
10 . . . . CONTINUE
```

he needs simply type

```
10  . CONTINUE
```

Lines typed to the editor may be changed if an error is made, using the same characters " and ? as are used on DOS command lines. The erase character " will cause the preceding letter to be deleted from the line. Successive use of this character will cause successive characters to be removed. A ? will "kill" the input line up to the ?. The erase and kill characters are effective whether in the INPUT or EDIT mode. The erase or kill character may be changed by means of the ERASE or KILL edit command.

The character ^ carat is the logical escape with the following conventions

^OO	hex character
^L	move to lower case
^L	move to upper case

Each of the editor commands may be abbreviated, as shown in each command.

An error which takes control out of the editor may be recovered by starting at 5A5E hexadecimal. This allows the user to reenter the editor with the text buffer the user was working with intact. The editor brings the entire file into core for editing. If the file is too large, the editor will type BUFFER OVERFLOW

LOADING. The editor can process a file up to about 2000 FORTRAN statements. If the file is too large, the user must use the SUPDATE command (3.2.9) to make modifications to the file. It might be useful to use SUPDATE to break the file into smaller pieces and use to editor to modify the pieces.

Following are the editor commands

Model:	TOP
Abbreviation:	T

TOP positions the pointer at the first line of the file.

Model:	BOTTOM
Abbreviation:	B

BOTTOM positions the pointer at the last line of the file, a dummy "null" line.

Model:	NEXT [*n*]
Abbreviation:	N[*n*]
Examples:	NEXT 5
	N -10
	N

NEXT moves the pointer forward *n* lines if *n* is a positive integer or backward *n* lines if *n* is a negative integer. If *n* is 0 or omitted, the editor assumes 1 and moves the pointer forward one line. If the end of the file is reached during execution of this command, the pointer remains at the last line and the editor types

```
    BOTTOM
```

If the beginning of the file is reached, the pointer remains at the first line and the editor types

    TOP

Model:	PRINT [*n*]
Abbreviation:	P[*n*]
Examples:	PRINT
	P 6
	PRINT -6

If *n* is a positive integer, PRINT types *n* lines on the teletype, beginning with the current line. If *n* is 0 or omitted, the current line is typed. If *n* is a negative integer, the pointer is moved *n* lines backward, and that line is typed. The pointer is left pointing to the last line typed.

If the beginning or the end of the file is reached during execution of this command, the editor types TOP or BOTTOM, respectively.

Model: *

An asterisk (*) can be specified as the last command of a command line to set up a loop consisting of all the commands in the line. The commands are then executed in the loop until either the beginning or end of file is reached. Thus, for example, the command line

```
NEXT, PRINT, *
```

will list a file from the current line through the last line. To list the entire file, type.

```
T
N, P, *
```

Model: RETYPE *line*

Abbreviation: R *line*

Examples:
```
RETYPE      GO TO 100
R\IF(ERRVEC(3).EQ.12)  GO TO 9
```

RETYPE replaces the current line with *line* and leaves the pointer positioned at the current line (i.e., *line*). Note that R and *line* are separated by exactly one blank; additional blanks are part of *line*.

Model: DELETE$\left\{\begin{matrix}[n] \\ \text{TO 'string'}\end{matrix}\right\}$

Abbreviation: D$\left\{\begin{matrix}[n] \\ \text{TO 'string'}\end{matrix}\right\}$

Examples:
```
DELETE 5
D
DELETE TO 'AB4'
```

If *n* is a positive integer, DELETE deletes *n* lines from the file, beginning with the current line. If *n* is 0, the current line is deleted. If *n* is a negative integer, the command is ignored. DELETE TO 'string' deletes up to, but not including, the line containing 'string'. The pointer is positioned at the line which follows the last deleted line. If the end of the file is reached during execution of this command, the pointer remains at the last line of the file, and the editor types BOTTOM.

Model: LOCATE *c*

Abbreviation: L *c*

Examples:
```
LOCATE GO TO 7
L CALL BRWFIL
```

LOCATE positions the pointer at the first line after the current line which contains the character string c anywhere in the line, where c can be a string of any length. If the end of the file is reached during execution of this command, the pointer remains at the last line of the file, and the editor types BOTTOM.

Model:	FIND c
Abbreviation:	F c
Examples:	F 100
	FIND 950

FIND positions the pointer at the first line after the current line which begins with the character string c, where c can be a string of any length. If the end of the file is reached during execution of the command, the pointer remains at the last line of the file, and the editor types BOTTOM. FIND is particularly useful for retrieving labeled statement lines or error messages in a compilation listing file.

Model:	CHANGE xc_1xc_2x [n][G]
Abbreviation:	C xc_1xc_2x [n][G]
	C /100/20/
	C AZA9A G

CHANGE examines n lines, beginning with the current line, where n is a positive integer or 0. If n is 0 or omitted, only the current line is examined. CHANGE replaces character string c_1 with character string c_2, where c_1 and c_2 can be strings of any length and need not be the same length. If G is specified, every occurrence of c_1 on a line will be replaced; if not, only the first c_1 on a line will be replaced. The character strings c_1

and c_2 are delimited by *x*, a character which you specify as the first nonblank character after CHANGE or its abbreviation. This character must not occur in c_1 or c_2.

The pointer is repositioned to point to the last line which was examined. If the end of the file is reached during execution of the command, the pointer remains at the last line of the file, and the editor types BOTTOM.

Model:	INSERT *line*
Abbreviation:	I *line*
Examples:	INSERT C THIS ROUTINE SEARCHES FOR A DELIMITER.
	I IF (B .NE. 100.0)

INSERT inserts *line* directly after the current line and repositions the pointer to point to *line*. Note that I and *line* are separated by exactly one blank; additional blanks are part of *line*.

Model:	ERASE *c*
Abbreviation:	E *c*
Examples:	ERASE :
	E /

ERASE changes the current erase character to *c*, any ASCII character which can be printed. The erase character, when typed, deletes the last character which was typed. If *c* is the current erase or kill character, comma (,), semicolon (;), plus (+), minus (-), or new line (carriage return or line feed), the command is ignored. The initial erase character is the quotation mark (").

Model:	KILL *c*
Abbreviation:	K *c*
Examples:	KILL X
	K ?

KILL changes the current kill character to *c*, any ASCII character which can be printed. The kill character, when typed, deletes the current line. If *c* is the current erase or kill character, comma (,), semicolon (;), plus (+), minus (-), or new line (carriage return or line feed), the command is ignored. The initial kill character is the question mark (?).

Model:	TABSET *x* [*x*] ...
Abbreviation:	TA
Example:	TABSET 20 40 60

TABSET sets tabs at each *x*, where *x* is an integer 1-72. Up to eight *x*'s can be specified, and they must be in ascending order. Tabs are set initially to 6, 12, and 30. To tab on input, type a backslash (\), not a tab character. For example, to input the following:

```
                A     B     C
```

type:

```
                A\B\C↵
```

Model:	PTABSET *x* [*x*] ...
Abbreviation:	PT
Example:	PTABSET 6 12 18

PTABSET, with the same argument format as TABSET, allows the user to tell the editor where the physical tab stops are on the teletype. EDIT will output TAB rather then spaces on output where appropriate. Initially tabs are set to 6, 11, 21, 31, and 41.

Model: BRIEF

Abbreviation BR

BRIEF suppresses typing of text lines for all but PRINT commands. When the editor is invoked, it will initially be in this condition.

Model: VERIFY

Abbreviation: V

VERIFY causes CHANGE, NEXT, FIND, LOCATE, and RETYPE commands to type out the current line after any changes have been made.

Model: INPUT

Abbreviation: IN

INPUT puts the editor in input mode. In response, the editor types INPUT.

Model: LOAD *file*

Abbreviation LO

Example: LOAD SCHRS

LOAD loads *file* into the editor buffer after the current line and puts the editor into edit mode. In response, the editor types EDIT. The named file must be listed in the current directory. After the file is loaded, the text pointer points to a null line following the loaded file.

Model:	FILE [*file*]
Abbreviation:	FILE INB
	FILE

FILE checks for *file* in the current directory and, if found, replaces the old file with the new edited version. If *file* is not found in the directory, an entry for *file* is created in the directory and the file is entered. If *file* is omitted and you have invoked the editor with the *file* specification, the file will be stored with that *file*. If you did not specify a *file* when invoking the editor and you omit a *file* in the FILE command, the editor will request a *file*.

After a FILE command is executed, control automatically returns to DOS. In response, DOS types OK,.

Model:	QUIT
Abbreviation:	Q

QUIT returns you to DOS. In response, DOS types OK,.

3.2.4 MEDIA: File-Copy Utility

Model: MEDIA

Examples: In all of these examples, system questions are underlined, user responses are not. Each user response is assumed to terminate with a carriage return.

Cards to Magnetic Tape

```
OK, MEDIA
INP MODE=S
```

```
INP=C
OUT=M
DRIVE=1
RECL=30
REWIND? Y
INP MODE=Q
```

Cards to Several Files

```
OK, MEDIA
INP MODE=S
INP=C
OUT=F
SEVERAL FILES? Y
OU=DS1
OU=DS2
OU=DS3
OU=↵
INP MODE=Q
```

Magnetic Tape to File

```
OK, MEDIA
INP MODE=S
INP=M
RECL=↵
DRIVE=0
SKIP(F)=0
SKIP(G)=0
COPY F OR G (ONE) F
OUT=F
```

```
OU=DSK5
SEVERAL FILES? N
SEPARATE GROUPS? N
REWIND INP TAPE? Y
INP MODE=Q
```

File to Print

```
OK, MEDIA
INP MODE=S
INP=F
SEVERAL FILES? N
OUT=L
CC CHARS? Y
IN=DS2
INP MODE=Q
```

MEDIA is a fairly general utility for copying files of data. It is capable of reading from any of the following devices:

device	*code*
file system (disk)	F
card reader	C
magnetic tape	M

and of writing to any of the following devices

device	*code*
file system (disk)	F
line printer	L
magnetic tape	M

MEDIA is capable of dealing with both files and groups. The end of a group is always indicated by a single end-of-group record, while the end of a file is indicated by two consecutive end-of-group records written as follows:

medium	*end-of-group record*
file system (disk)	record with $EOF in position 1-4
card reader	card with $EOF in columns 1-4
line printer	top of page
magnetic tape	tape mark (standard) or record with $EOF in position 1-4 (input only)

End-of-file on the disk is determined by exhausting the data count in the last record of the record chain (or by encountering two consecutive end-of-group records, if that should occur).

When either one of the devices participating in the data transfer is the disk (file system), it is possible to associate groups on one side with files on the other. That is, if one answers Y to the question SEVERAL FILES? (see below), MEDIA will treat end-of-group on the input device as end-of-file on the disk, or end-of-file on the disk as end-of-group on the output device. For example, this allows one to print several files or to create several disk files from a multi-group magnetic-tape file without repeating all of the setup commands. If one is copying from disk to disk and has answered twice with Y, MEDIA will copy several input files into an identical number of output files.

Flow of Control: Setup for Operation and Termination

MEDIA operates in either source mode (ASCII) or object mode (binary). It does not convert from one to the other. When MEDIA is invoked, and after it completes any copying operation, it asks

```
INP MODE=
```

You must respond to this with S or O, for source or object, respectively, or with Q to return to DOS.

Having determined the data mode, MEDIA will ask for an input medium and an output medium, together with any supporting information required. This sequence proceeds as follows. MEDIA types:

```
INP=
```

to which you respond with F,C, or M (see the list of codes above). If any further information is required about the input source, such as drive number for magnetic tape, MEDIA now asks for it as indicated in the device-specific sections below. MEDIA then types

```
OUT=
```

to which you respond with F,L, or M (see the list of codes above). Again, if further information is required, MEDIA requests it. MEDIA then proceeds to copy your data according to your specifications. If either your input device or your output device is the file system, MEDIA will ask for one or more file names by typing

```
IN=     or     OU=
```

to which you must respond with a filename. Depending on the options you have chosen, you may be asked for more than one name for either input or output files. Typing carriage return for an input file will terminate the current operation, while typing carriage return for an output file will cause one group from the input file to be skipped.

Device-Specific Considerations

Depending on your choice of input and output devices, MEDIA may require some elaboration. This is described below on a device-by-device basis.

<u>File System</u>

SEVERAL FILES? expects a Y or N response and effectively asks whether you wish to continue the operation with another file when finished with the first. For example, to list several files, you would set up just once and then simply be asked to specify a new input file (IN=) when the last one specified is exhausted. This sequence terminates when you reply to IN= with only a carriage return.

SEPARATE GROUPS? also expects a Y or N response, but is <u>only asked if you are specifying file system output</u>. This allows you to write end-of-group records or not, as you wish. Typically, you should respond Y in source mode and N in object mode (for example, FTN expects group separators, but LDR does not).

As indicated earlier, after you have completed your input and output specifications, you will be asked to identify one or more input files (IN=) or one or more output files (OU=).

<u>Card Reader</u>

No additional information is needed.

<u>Line Printer</u>

CC CHARS? expects a Y or N answer and is asked only for source mode. If you specify Y, the first word of each record will be used as a carriage-control word (see DOPRIN, 4.6.3), and will not be printed. If you specify N, MEDIA will print 50 lines per page with an appropriate heading that incorporates the first record in the current group. When processing several groups, MEDIA will begin each group on a new page.

When operating in object mode, MEDIA always provides page control. Records are printed eight words per line with double spacing between records. If input is from disk, the response typed for IN= is used for a title. Otherwise, MEDIA asks TITLE=.

Magnetic Tape (Input and Output)

DRIVE= asks you to specify a drive; valid responses are 0,1, 2,3.

RECL allows you to specify a record length (in words). If you simply type a carriage return, the actual record length (maximum 100 words) is used for input, and the input record length is used for output. The latter is determined as follows: file system source, 35; file system object 20; cards 20; magnetic tape, as specified. If you specify a record length, records will be truncated or extended with blanks (source) or zeroes (binary) as appropriate. You may specify a record length for input or output or both.

At the end of the operation, you will be asked whether you wish to REWIND INP TAPE, to which you should answer Y or N.

Current Media Problems

a. Although SEVERAL GROUPS? works in either direction, it may not work in both directions together.

b. When copying from the card reader, using several groups, MEDIA may get confused at the end. If it asks for too many filenames, abort MEDIA using the BREAK key on the A command. You cannot specify different responses for input and output when asked RECL=.

3.2.5 BEDIT: Manipulating and Inspecting Files of Binary Programs

Model: BEDIT

BEDIT invokes the binary edit program which allows inspection and manipulation of object-program files. Currently, the most important function of BEDIT is the maintenance of the file LIBRARY in LIBDIR, which is used by the loader for resolving references to library routines. An object-program file is a file of one or more object programs which are not separated by end-of-group records.

When BEDIT is invoked, it asks for the name of an output file by typing

```
OUTPUT FILE =
```

to which you respond by typing the name of a file and a carriage return. If you want to return to DOS, type QUIT.

If a filename or a blank line has been entered, BEDIT will request an input file by typing

```
IN =
```

to which you respond by typing the name of a file and a carriage return. If you want to close the current output file, if any, and open a new output file, just hit a carriage return without entering a name. BEDIT will respond by requesting a new output file.

If you have entered an input file name, BEDIT will request an operation command by typing >. Your response must be a command line of the following format:

command [*parameter*] ...

where each *command* is one of the commands described below. When you hit a carriage return, BEDIT begins to process the commands specified in the line. When a command line has been processed, BEDIT requests another line by typing >.

Model:	T [*n*]
Examples:	T
	T 5

T transfer *n* programs from the input file to the output file, where *n* is an integer 1-500. If *n* is omitted, the rest of the programs in the input file or 500 programs, whichever is less, are copied. If the end of the input file is reached, the rest of the commands on the line are ignored, and a new input file is requested.

Model:	S [*n*]
Examples:	S 1
	S 10

S skips *n* programs in the input file, where *n* is integer 1-500. If *n* is omitted, BEDIT closes the input file and requests a new one.

If the end of the input file is reached during execution of S, the rest of the commands on the line are ignored, and a new input file is requested.

Model:	B [*n*]
Examples:	B 5
	B

B backs up *n* programs in the input file, where *n* is an integer 1-500. If *n* is omitted, 1 is assumed. If the file is positioned fewer than *n* programs from the beginning, the file will be backed up to the beginning.

Model:	E [*n*]
Examples:	E 1
	E 15

E lists the entry points of *n* programs in the input file, where *n* is an integer 1-500. If *n* is omitted, the remainder of the file or 500, whichever is less, is assumed. Entry points are listed one per line at the left-hand margin of the page, with a blank line between programs.

Model:	R [*n*]
Examples:	R
	R 5

R lists the entry points and external references of *n* programs in the input file, where *n* is an integer 1-500. If *n* is omitted, the remainder of the file or 500, whichever is less, is assumed. Entry points and external references are listed one per line; entry points are typed at the left-hand margin of the page, and references are typed in a column indented beneath the appropriate entry point. Program listings are separated with a blank line.

Model: Q

Q returns to DOS, which responds by typing OK,. This works only if BEDIT has just typed >. If BEDIT has typed OUTPUT FILE=, type a carriage return. If BEDIT has just typed IN=, type the name of any file in the current UFD; BEDIT will type >.

Model: I

The I (INSERT) command will open a second file for reading and copy it entirely. Upon completion, the second file is closed. During an I, the binary location pointer is unmoved on the original input file. After I is typed, BEDIT responds INSERT FILE=. User types the name of a file. The entire file is copied onto the output file

Model: N

The N command, indicating NEW INPUT FILE, causes BEDIT to close the current input file and ask for the name of a new one by typing "IN=". The user responds with the name of a new binary file. This command is useful for extracting programs from several libraries and combining them into a new library.

Model: O

The O (OUTPUT) command will close the current output file and open a new file for writing. When O is typed, BEDIT responds "OUTPUT FILE=". User types the name of a new file. The O command is useful for breaking long binary files into shorter ones, each containing one or more subroutines.

Model: K

This special command allows the user to add routines to an existing file. When BEDIT is invoked, the user specifies the name of an existing file when OUTPUT FILE= is typed. The K command is given as the first command to BEDIT and causes BEDIT to skip to the end of the output file. Subsequent T or I commands will append routines to the output file.

3.2.6 LDR: Loading a File

Model: LDR

LDR invokes the loader to load a file which contains one or more object programs (see CONCAT). An object program is generated by the compiler or the assembler from a source program. It is not executable because it contains relative addresses and unresolved external references. That is, references to locations within the program itself (e.g., statement labels) are expressed in terms of their relationship to the beginning of the particular program or one of its associated storage areas, and references to other programs (e.g., calls to system I/O routines) are left blank and are flagged. Before a program can be executed, the loader must convert each of these references into an actual core address. This involves assigning specific core locations to the program and to any other programs which are to be executed with it.

After an object program has been loaded into core by LDR, you can create a copy of it by requesting the LDR save function or by using the SAVE command (3.4.1). Such a file is usually referred to as a load module. Under DOS, you can give this type of file a name and save it by entering its name in your directory. A load module is executable and need not be processed by the loader each time you want to execute it.

When the loader has assumed control, it types ">" to request a control line consisting of one or more control parameters, separated by commas and terminated by a carriage return. Three types of control line are accepted:

1) Normal load line
2) Force load line
3) Save control line

<u>Normal Load Line</u>

A normal load line is supplied in the following general form:

$$\left\{\begin{matrix}*,\\ \$a,\\ 0,\end{matrix}\right\} \text{PRI},\textit{file} \begin{bmatrix},\text{REF}\\ ,\text{REFN}\end{bmatrix} \begin{bmatrix},\text{OBJ}\\ ,\text{OBJN}\end{bmatrix} \begin{bmatrix},\text{MAP}\\ ,\text{MAPN}\end{bmatrix} \begin{bmatrix},\text{COM}\\ ,\text{COMN}\end{bmatrix} \begin{bmatrix},\text{LIB}\\ ,\text{LIBN}\end{bmatrix}$$

The first parameter of a normal load line is the load parameter which determines the address at which loading begins. It may be specified as one of the following:

1)	*	asterisk
2)	$*a*$	where *a* is 1-5 hexadecimal digits
3)	0	numeric zero

"*" sets the load address at the first even location not previously used during the current execution of the loader. If no previous loading has occurred, a default location above the loader is selected. "$*a*" specifies an explicit hexadecimal load address. "0" defines a dummy load line. No loading takes place but the remainder of the line is scanned for other parameters.

An explicit hexadecimal value on the first load line may specify an address within the loader itself. In this case, the program will be adjusted for execution at the load address, but will be loaded into the area above the loader. Once established this "offset load" mode remains in effect for subsequent load lines. When loading is successfully completed, the program is moved down to the correct location for execution. Control is then passed to DOS.

The default location to load programs is 6400 hex. The user cannot specify a load address less than 4F00 hex, as the operating system occupies that space. The loader occupies locations 4F00 to 63FE hex.

Other parameters which may appear on a normal load line are:

PRI,*file*

PRI,*file* specifies the name (up to eight characters) of the primary file to be loaded. The *file* must exist in the currently attached directory.

The remaining six pairs of parameters specify loader processing options. The initial default value is underlined. Each specification continues in effect across load lines until explicitly respecified.

REF/<u>REFN</u>

REF specifies that processing is to be continued even if unresolved external references remain after processing the current load line. REFN terminates loading if unresolved references remain.

OBJ/OBJN

OBJ specifies that processing is to be continued if an object text error is encountered. (The object record is discarded.) OBJN terminates loading if an object text error is found.

LIB/LIBN

LIB specifies that the system library (LIBDIR) LIBRARY is to be searched to resolve any unsatisfied references remaining after loading primary file specified in the same load line. LIBN disables the search.

MAP/MAPN

MAP specifies that a symbol table of external names be printed on the teletype and on the line printer after the current load line has been processed. Symbols internal to DOS are printed but not typed. Symbol definition typing can be disabled by setting sense switch A (type "BREAK", "S", then "A"). MAPN suppresses symbol table printing and typing.

COM/COMN

COM specifies that blank common be allocated immediately following the end of the first subprogram loaded. COMN postpones allocation of blank common until the current load line has been processed.

Force Load Line

The force load line causes specified external names to be added to the loader's symbol table. Two formats are allowed:

1) *file, load,name1,...,namen*

2) *NO,name1,...,namen*

If specified, *file* must be a file in the currently attached directory. The names are first added to the symbol table as unsatisfied external references, and then the file *filename* is searched in the same manner as the system library search requested by LIB. The load parameter is specified as in normal load lines by "*" or "*$nnnn*" ("0" is not permitted). "NO" causes the names to be added to the symbol table but no search is performed.

<u>Save Control Line</u>

After some or all of the object text has been loaded into core, the SAVE control line can be used to create a save file (an executable program) in the currently attached directory. The format of the line is:

SAVE,*file*,*sa*,*ea*,*epa*

The *file* is used to open a file in the currently attached directory using unit 2. Addresses *sa*,*ea*, and *epa* specify (without leading $) the starting, ending and entry point addresses for the save file in hexadecimal.

If the program has been loaded in "offset load mode", an "offset save" is performed. Thus *sa*,*ea* are the relocated values, i.e., the addresses used to load and execute the program.

Following processing of the SAVE control line, the loader requests another control line. Additional object text may be loaded or another save may be requested.

<u>Notes on Using the Loader</u>

With the exception of the system library, all files identified to the loader are presumed to reside in the currently attached directory. If the name of a specified file (primary or force load) is not found in the currently attached directory, the loader will type:

name NOT IN UFD. TYPE NEW NAME OR CR.

The user may retype a misspelled name and proceed. If a carriage return is typed, the loader will exit to DOS which responds with "OK,". The user may then type:

```
     ATTACH directory,START
```

The loader will then continue processing. Further filename references will search the newly attached directory.

Following the processing of each control line, the loader will request another load line except if unresolved external references remain and the REFN option is in effect, in which case the loader returns to DOS. If multiple load lines will be required to load a program, the REF option should be specified in the first line. To return to DOS, type a carriage return in response to the next control line request.

Since the table of external references is retained throughout a single execution of the loader, unresolved references which remain after processing a load line can be resolved by specifying the necessary files in the PRI or force load specification in subsequent control lines. After all necessary subprograms have been loaded, the program may be saved and/or started. The use of the SAVE command line is recommended since it provides printed documentation of the save filename, core occupancy, and entry point on the load map.

References to entry points within the resident DOS are resolved by searching a list of system names after each load line has been processed. The list is internal to the loader. It contains entry names only for routines to which the user should have access. Caution should be exercised to prevent the use of DOS entry names as external names in user written programs. Multiply defined external references may lead to improper operation of the loaded programs. See Appendix F for a listing of DOS system routines.

If the program to be loaded is larger than the available area of core, it may be possible to restructure some of the routines to permit overlay operation. The procedure used to build an overlay structure is described in Appendix G.

Examples of Loading Programs

```
user:      LDR
response:  >
```

The loader is invoked and awaits a control line

```
user:      *, PRI, B:PROG, LIB,MAP
response:  >
```

The file B:PROG in the current file directory is loaded starting at the default load address 6400 hex. The file LIBRARY in LIBDIR is searched to satisfy unresolved references, then the loader resolves any references to DOS routines through use of an internal table. The load map is printed and typed. If pseudo sense switch A is set, the load map is only printed.

```
user:      SAVE,*PROG,6400,7900,6450
response:  >
```

The core image file *PROG is generated containing locations 6400 to 7900 with entry point 6450. To run the program, user types RESUME *PROG. The first location to be saved is the first location loaded, normally 6400. The last location is found by referring to the address following FUL (first unused location) in the load map. The entry point is the address following MAIN in the load map.

```
user:       carriage return
response:   OK
```

User types a blank line to return to DOS.

```
user:       LDR
response:   >
user:       *, PRI, B:PROG1, REF
response:   >
```

Load B:PROG1, allow unresolved references, resolve any references to DOS routines.

```
user:       *, PRI, B:SUBR, LIB, MAP
response:   >
```

Load B:SUBR following B:PROG, search the library and generate a load map.

```
user:       SAVE, *PROG1,6400,8400,6450
response:   >
user:       carriage return
response:   OK
```

generate core image file as in first example.

```
user        LDR
response:   >
user:       $4F00, PRI, B:PROG, MAP, LIB
response:   >
```

Load program starting at 6400 with addresses relocated as if it were loaded starting at 4F00. Search the library and resolve references to DOS. Print and type the load map as if the load was made at 4F00.

```
user:      SAVE, *PROG, 4F00, 5F00, 4FF0
response:  >
```

Generate the core image file *PROG in an "offset save". The parameters given to the SAVE command are the relocated values the addresses used to load and execute the program. The loader carefully generates the file with the text from locations 6400 to 7400 in this case.

```
user:      carriage return
response:  OK
```

The entire load is moved into the correct area overlaying the loader, then control is returned to DOS. The user could then give the DOS SAVE command instead of the loader SAVE command as follows:

```
user:      SAVE *PROG 4F00 5F00 4F00
response:  OK
```

The correctly located load of the program is saved from 4F00 to 5F00 with entry point 4FF0. Following are some of the error messages which the loader issues.

ALLOCATION ERROR *xxxxxxxx*

Common request appeared in midst of program. *xxxxxxxx* is COMMON name.

BAD HEX NUMBER

Error in hexadecimal number in load line. Loader returns to read a new line.

BLANK COMMON ERROR *xxxxxxxx yyyyyyyy*

Large blank common request appeared in data stream after common allocated. *xxxxxxxx* is allocated size; *yyyyyyyy* is new size.

$$\text{CHECKSUM ERROR ON} \begin{Bmatrix} \text{LIBRARY} \\ \text{MAIN FILE} \\ \textit{file} \end{Bmatrix}$$

The loader got a checksum error while trying to load from a file. If the error is on the library, contact a systems programmer. If the file is your own and you decide, after reconsideration, that it is not a binary file but you do have another file to load, type:

```
CLOSE 1
ATTACH directory
INPUT file
START
```

and the loader will resume with your new program. If you are sure that your file is a binary file, the utility DOSUM will correct the checksums in the file. But be careful - any text, data, or core image file fed to DOSUM will be hopelessly and irretrievably ruined.

FILE NOT PROPERLY OPEN

You did not specify a PRI file.

FILE *file* NOT IN UFD. TYPE NEW NAME OR CR (bell rings)

A file you specified was not in the current UFD. If you meant another file in your UFD (e.g., if you misspelled it), type the name and carriage return. If you meant a file of that name but were attached to the wrong UFD, carriage return and you will be back in DOS. Then type:

```
ATTACH directory
START
```

ILLEGAL LOAD ADDRESS *x xxxxxxxx*

ILLEGAL STRT ADDRESS *x xxxxxxxx*

Currently these errors are not caught, but the tests will eventually be implemented.

MEMORY OVERFLOW 0001 *xxxxxxxx*

Load conflict with symbol table. *xxxxxxxx* is address of end of symbol table. Terminal error.

MEMORY OVERFLOW 0002 *xxxxxxxx yyyyyyyy*

TEM or COM conflict with symbol table. *xxxxxxxx* is address of end of table if allocation could be made; *yyyyyyyy* is address of end of load. Terminal error.

MEMORY OVERFLOW 0003 *xxxxxxxx yyyyyyyy*

Insufficient room in activity area. *xxxxxxxx* is address of end of area for TEM or COM allocation if allocation could be made; *yyyyyyyy* is address of end of load. Terminal error.

NO UFD FOR *file*

No UFD was attached. Type:

```
ATTACH directory
START
```

OBJECT ERROR IN *xxxxxxxx yyyy*

Messed up binary file or nonbinary file; *xxxxxxxx* is program name, and *yyyy* is hexadecimal record number. If OBJ condition is in effect, loader reads next record; if not, it types > and waits for a new load line.

PARAM ERROR *xxxx*

xxxx is the hexadecimal sequence number of an incomprehensible specification in the load line. Loader returns to read a new line. If you can discover no error, this message may indicate that the teletype is malfunctioning. Just in case, try the line again, and, if you still have trouble, try the TELETEST program.

REFS *xxxx*

xxxx is number of unresolved references.

UNDEFINED NAME IN LOCSET *xxxxxxxx*

xxxxxxxx is the undefined name. This message should occur only for MAC programs.

3.2.7 BUGGY: Debugging a Program On-Line

Model: BUGGY

BUGGY invokes the on-line debug package that was developed to help check out DOS. BUGGY allows you to examine the contents of any address, change those contents, perform address computation, and insert and remove break-points. BUGGY is a part of DOS; it therefore does not take up any space in the user section of core memory.

When BUGGY takes control, it saves the contents of all registers and types:

```
BUGS
address contents
```

where *address* is the location in which the *contents* of register 0 have been stored. If you want the same information for register 1, hit a carriage return or line feed (↵), and BUGGY will type the address at which register 1 has been stored and the contents of register 1, in the same format. Each time you hit a carriage return or a line feed, BUGGY will go on to list the *address* and *contents* of the next register.

In general, BUGGY types out numbers in hexadecimal representation and interprets all numbers that you type as hexadecimal numbers. If you want to express a number in decimal, precede it with a period. The examples in this section show numbers of both types.

After BUGGY types out the storage location and contents of a register, you can change the contents of a register by typing in the new contents. In the following example, the contents of register 0 are changed from 491E to 7A1:

```
BUGS
00003960 0000491E 7A1
```

Be sure not to space before typing the replacement value because inserting a space takes BUGGY out of the alter mode, and then any number that you type will not replace the contents of the most recently displayed register. The number you type will replace the contents of the current register only if you (1) do not precede it with a space and (2) immediately follow it with either a carriage return, line feed, or space. If you follow it with any other command (e.g., L, see below), the number will be associated with that other command and will not modify the contents of the register.

It is possible to examine selected registers without having to wait for BUGGY to list them in sequence. If you want to see the location and contents of registers 0, and 8, respond as follows:

```
BUGS
00003960 0000491E↵
00004396 946273A R8L↵
```

That is, type R, the register number, and L (for List). You don't have to space before R in this case because the number is directly followed by, and thus associated with, another command. An alternative is to tell BUGGY to go ahead eight words or sixteen half-words, by typing one of the following:

```
*+8WL
*+10L (or, in decimal, *+.16L)
```

where * is the current address and W designates words.

You can use these same techniques of displaying and changing contents for any address. Suppose, for example, you have seen and modified all the registers in which you are interested, and now you want to examine other locations. You would type the address followed by an L and BUGGY then would display that address and its contents. If you want to change those contents, you would type a number followed by a space. If you wanted to go on to another location, you would type the new address followed by an L. Hitting a carriage return or line feed displays the contents of the next word. In the following sequence, these features are demonstrated:

```
0000491E 061FAB61 0600AB62 5BAL↵
000005BA 0000FFFF↵
000005BC 00001092
```

You can back up a location by typing <, which is the equivalent of specifying *-2L or *-1WL.

Other BUGGY commands and options are presented below.

Model: ↑ ↑

BUGGY provides indirect addressing by treating the contents of a location as an address. If you type ↑, BUGGY will treat bits 15-30 of the last number it typed as an address, go to that location, and display its contents, as in the following sequence:

```
BUGS
00003960 6F02491E
0000491E 061FAB61
```

Model: /

If you want to change the contents of a location, but use the original contents as the address of another location which you want to examine, type a slash (/) followed by a command, as demonstrated in the following sequence:

```
0000491E 061FAB61 369F /L↵
0000AB61 238E0021
```

No matter where you are in a line, i.e., regardless of the number of things you have already typed, / gives you access to the *contents* of the location which BUGGY typed out at the beginning of the line.

Model: *address* [G]

To initiate execution of a routine, depress the CNTRL key and type G (represented in the Model above as a bracketed G). BUGGY will go to *address* and begin execution.

Model: *address* [B]

[R]

To insert a breakpoint in your routine at *address*, depress the CNTRL key and type B (represented in the model above as a bracketed B). When the routine is being executed, it will break at *address* and pass control to you. The instruction at *address* will not have been executed when the break occurs. You can do whatever you like when control is in your hands; to resume execution with the instruction at *address*, depress the CNTRL key and type R (represented in the Model above as a bracketed R). When the instruction at which a breakpoint has been inserted is executed, it will be executed out of line; caution must therefore be exercised regarding which instruction has been affected (e.g., the LNJ command).

BUGGY allows only one breakpoint at a time. If you insert several, all but the last will be ignored. When execution of the routine breaks and control passes to you, you can change the breakpoint by typing:

address [B]

Typing [R] in this case will still return you to the instruction at which the original breakpoint was inserted. To remove a breakpoint without inserting another, type the following:

0[B]

Model: *arithmetic expression*

BUGGY will evaluate arithmetic expressions. If you have typed an expression and want to know the result, just type an equals sign (=), as in the following:

```
94F261+BF+39762-7A1=
```

BUGGY will type out the result in hexadecimal immediately following the equals sign and will allow you to continue typing expressions using that result. If you space after a result is typed out, you can begin a new expression.

If you want the result expressed in decimal, type a colon (:) instead of an equals sign. When doing decimal arithmetic, BUGGY does not treat numbers as being in integer position, e.g., a FORTRAN 4 will be considered and typed as 8.

Model:
```
?
```

If you make a mistake, type ?. It will kill the current line. If you type something which BUGGY does not recognize, it will type ? and restart the current line.

Model   Q

To return to DOS, type Q. DOS will respond by typing OK,.

3.2.8 CONCAT: Preparing Input for the Loader

Model: CONCAT

Example:
```
OK, CONCAT
OUTPUT FILE=C:LLDX
NAME FILE=N:LLDX
NAME FILE=Q
OK,
```

Because of the DOS facilities for file management, it is generally more convenient to maintain subroutines separately. Thus, instead of always compiling all of the subroutines in a program, one need compile only those which are being changed. The rest can be kept in the UFD, which acts as a library. If this is done, however, it is useful to have some facility which will collect several subroutines together so that they can be processed by the loader. In lieu of such a facility, you must specify the routines individually to the loader via multiple command lines.

CONCAT allows the user to create and reuse a list of required subroutines. Operating with this list, CONCAT copies the named files into a single output file which is acceptable to the loader. Thus the user can list his subroutines once, modifying the list only as necessary, and then use the list repeatedly as a basis for subsequent loads.

When invoked, CONCAT types:

```
        OUTPUT FILE=
```

In response to this request, you should specify a filename, for example,

```
        (TSTONE SECRET) PAPERS
```

CONCAT then types=

```
        NAME FILE=
```

Now you should specify a second filename. CONCAT will then copy the files named in the name file into the output file. When this has been accomplished, it will again type:

```
        NAME FILE=
```

You may either supply the name of another list of files or type Q or QUIT, thus returning control to DOS. You may specify as many name files as you wish. As is conventional for DOS object files, no end-of-group records are written in the concatenated output file.

A name file consists of one or more lines, each containing one or more filenames. Directory names are enclosed in parentheses as usual. When no directory name is specified with the filename, the last directory mentioned is used. For example:

```
        (AXEL) LIST SUBJ
```

indicates two files LIST and SUBJ, both from directory AXEL. If no directory at all is specified, the current directory is used. Empty parentheses can also be used to specify the current directory. For example, the following line:

```
        (AXEL) LIST SUBJ () HELA
```

will cause HELA to be copied from the current directory after LIST and SUBJ have been copied from AXEL.

3.2.9 SUPDATE: Modifying a Source File

SUPDATE is a noninteractive alternative to EDIT. It differs from the EDIT in that all changes must be known before the beginning of the run. Furthermore, the process is controlled by sequence numbers appearing in columns 73-80 of both the source file and the control file (except for the DEL control card). SUPDATE can be used to modify files which are too large for EDIT. It can also be used when you want to keep a reusable copy of your changes (e.g., a card deck).

SUPDATE uses four disk files which should be OPENed (3.3.4) on the following units before starting

unit	*function*
1	old master - source to be modified (one group only)
2	control card file
3	new master - the result of UPDATing
4	listing file - essentially a listing of a new master and not of the control cards

Unit 1 may be omitted if column 29 of the CHG card contains a *. Units 2, 3 and 4 are required.

Both the old master and the control card file must be in ascending order by sequence number. Blanks in the sequence field are not equivalent to zeroes. Nonnumeric characters may be used in columns 73-75. UPDATE reads both the old master and the control card file and such operations as replacement of records, insertion of records, and deletion of records.

Since SUPDATE operates exclusively within the context of the DOS file system, one may have to use MEDIA (3.2.4) for certain functions. These might include copying the old master from tape to disk and creating or listing the control card file.

Use COPYFS to renumber a source file or add sequence numbers to a source file. See 3.6.5 for a description of COPYFS.

Control card formats are presented below.

<u>Operation Card</u> This card must be the first in an update deck. It must be punched as follows:

Column(s)	*Contents*
1-2	./
3	blank
4-6	CHG
7	blank

8-10	optional three-character code to be placed in columns 73-75 of all records in the new master
11	blank
12-16	optional five-digit initial sequence number
17	blank
18-22	optional five-digit increment for sequence numbers
23	blank
24-27	LIST if new master is to be listed; blank if not
28	blank
29	* if there is no old master and the new master is to be made up from Insert Records (file creation rather than file update); blank if an old master is being used.

<u>Replaced Record</u> Any number of these cards may be present in an update dack. Each card must be keypunched as follows:

Column(s)	*Contents*
1-72	Contents of new record
73-80	Sequence number of record to be replaced

<u>Insert Record</u> Any number of these cards may be present in an update deck. Each card must be keypunched as follows:

Column(s)	*Contents*
1-72	Contents of new record
73-80	Sequence number which falls between the sequence numbers of two consecutive records in the file, unless file creation is taking place (* in column 29 of CHG card), in which case sequence numbers on insert cards are optional

<u>Delete Range of Records</u> Any number of these cards may be present in update deck. Each card must be keypunched as follows:

Column(s)	*Contents*
1-2	./
3-4	blank
5-7	DEL
9-16	sequence number of first record to be deleted
17-24	sequence number of last record to be deleted (or blank if only one record is to be deleted)
25-80	blank

<u>Begin Printing</u> Any number of these cards may be present in an update deck. The LIST option on the operation card is overriden. Each card must be keypunched as follows:

Column(s)	*Contents*
1-2	./
3-4	blank
5-7	LON
8-72	blank
73-80	sequence number of record at which printing should begin

<u>Stop Printing</u> Any number of these cards may be present in an update deck. The LIST option on the operation card is overridden. Each card must be keypunched as follows:

Column(s)	*Contents*
1-2	./
3-4	blank
5-8	LOFF
9-72	blank
73-83	sequence number of record at which printing should begin

<u>Comment</u>	Comment cards are used to describe the modifications being made. Any number may be present in the update deck. They are ignored. Each of these cards must be keypunched as follows:

Column(s)	*Contents*
1-2	./
3-4	blank
5-7	COM
8	blank
9-80	any comment

3.3 I/O Control Commands

DOS provides a set of commands that control the activity of files when they function as input or output for other files. A file becomes active when you connect it to a unit. DOS currently provides you with 7 units (identified with an integer 1-7). Typically, a program performs all of its data transfers through a fixed set of units, with specific units associated with specific functions. This does not mean, however, that a user may process only a fixed set of files. At run-time the user is free to associate any appropriate file with each of the units employed by the program. For example, the FORTRAN compiler expects disk input on Unit 1 (see 3.2.1) and the user is free to open any FORTRAN source file on that unit. Similarly, if your own program contains the statement

```
          READ (7,1000) X, Y, X
```

you can assign any appropriate file to unit 7 and have it processed by your program. When a file is connected to a unit, it is open; when it is not connected to a unit, it is closed. In many cases you are responsible for all opening and closing of files. Some programs, however, ascertain the files to be used and open them themselves (e.g., DMPSAV).

Most of the DOS I/O control commands are associated with specific units. That is, the INPUT command (3.3.1) opens a file on unit 1, and the LISTING command (3.3.3) opens a file on unit 2. Other DOS commands, notably the system programs, are similarly associated with these units. Thus, the FORTRAN compiler (3.2.1) accepts as input the file which is already open on unit 1. This means that before invoking the compiler you must open a file on unit 1, and you will typically do so by issuing an INPUT command (unless your input is on cards). (You can alternately use an OPEN command [3.3.4], but in this case you would have to specify unit 1 explicitly.)

At any moment, a given file can be connected to only one unit, and a given unit can have only one file connected to it; DOS will flag an error condition if you try to open a file which is already open or to connect a file to a unit which already has a file connected to it. This can sometimes be confusing, so it is good to get into the habit of closing files after using them.

It is good programming practice to avoid the use of I/O control commands. Users should use subroutines OPEN and CLOSE (see section 4.3) to open and close files. User programs can request the name of a file to be opened for flexibility by typing a message and reading the name from the teletype. Subroutine GTNAMS (see 4.5.29) is useful in extracting the filename typed.

3.3.1 INPUT: Opening an Input File

Model:	INPUT [(*directory password*)] *file*
Examples:	INPUT MBED
	INPUT (TSTONE) BOPRIN

INPUT connects *file* to unit 1. If another *directory* is specified and the *passwords* match (currently blank), DOS assumes that *file* is located in *directory* rather than in the current directory.

3.3.2 BINARY: Opening an Object-Text File

Model:	BINARY [(*directory password*)] *file*
Examples:	BINARY B

BINARY opens unit 3 to write an object file. This file is written on disk as *file*. If the name of an alternate *directory* is specified and the *passwords* match (currently blank), the *file* is placed in that directory; if not, it is entered in the current directory.

3.3.3 LISTING: Opening a Listing File

Model:	LISTING [(*directory password*)] *file*
Examples:	LISTING L1

LISTING opens unit 2 to write a source listing. The file is written on disk as *file*. If the name of an alternate *directory* is specified and the *passwords* match (currently blank), the *file* is placed in that directory; if not, it is entered in the current directory.

3.3.4 OPEN: Opening Any File

Model: OPEN[(*directory password*)] *file unit key*

Examples: OPEN (KEEN) STOPEDL 5 1

OPEN opens the specified *unit* (integer 1-7) and associates with it the specified *file*. You specify a *key* which indicates the type of activity for which the file is being opened; specify a *key* of 1 for reading, 2 for writing, or 3 for both. If you include the name of another *directory* and the *passwords* match (currently blank), the *file* is associated with that directory rather than the current directory. Thus, if you open *file* for reading and you do not specify a *directory*, DOS will search the current directory for *file*; if you specify a *directory*, it will search that directory for *file*. If you open *file* for writing and you do not specify a *directory*, DOS will enter *file* in the current directory; if you specify a *directory*, it will enter *file* in that directory.

3.3.5 CLOSE: Closing a File or Unit

Model: CLOSE $\left\{\begin{matrix} \textit{file} \\ \textit{unit} \end{matrix}\right\}$ ...

Examples:

```
CLOSE 1
CLOSE 2 3
CLOSE SCHRSCE
CLOSE ALL
```

CLOSE closes the named *file(s)* or specified *unit*(s). CLOSE ALL closes all files and units.

3.3.6 COMINPUT: Switching Command Input from the Console to a File

Model: COMINPUT $\left\{\begin{array}{l} \textit{file} \\ \text{TTY} \\ \text{CONTINUE} \end{array}\right\}$

Examples:
```
COMINPUT INP

COMINPUT TTY
```

COMINPUT allows users to prepare a list of commands with the editor, file it on the disk, and have DOS read teletype input from this file rather than from the teletype. The command COMINPUT *file* causes DOS to take subsequent teletype input from *file*. The last command in *file* should be COMINPUT TTY, which tells DOS to take subsequent commands from the teletype. Example: Using the editor, a user creates a file PMLIST which consists of the lines:

```
PM
LISTF
COMINPUT TTY
```

When the user types

```
COMINP  PMLIST
```

DOS types back

```
OK, PM

PSW1 = num    PSW2 = num
RO-7 = num, ..., ...
R8-15 = num, ..., ...
OK, LISTF
List of user files typed out
OK, COMINP  TTY
OK,
```

To have DOS type the results of commands PM and LISTF, the user can type one command to DOS instead of two.

DOS reads input from *file* by opening unit 12, causing subroutine RDASR to pick up lines by reading from unit 12 rather than from the teletype. When the command COMINP TTY is encountered, DOS closes unit 12 and takes subsequent commands from the teletype.

Any DOS error message will cause command input to be switched to the teletype, but the command input file is left open. A user may retype the command that caused the error message then continue reading from the command input file by typing

```
          COMINPUT  CONTINUE
```

Do not use the command CLOSE ALL in a command input file. This will close the command input unit and cause the message "COMINP FILE CLOSED". If a user wishes to abort a sequence of commands, he should push the BREAK key, then type A. DOS will respond "COMINP FILE CLOSED".

COMINPUT affects all teletype input read through subroutine RDASR. It does not affect teletype input through subroutine T1IN. COMINPUT affects teletype input for all commands except MEDIA and EDIT. COMINPUT also affects teletype input using FORTRAN unit 100.

COMINP is useful for updating large programs which consist of many files. For example, suppose a user has a program consisting of three FORTRAN source files. To run, a user makes up the following command input file DPROG.

```
INPUT  MAIN
BINARY  B:MAIN
FTN
INPUT  SUB1
BINARY  B:SUB1
FTN
INPUT  SUB2
BINARY B:SUB2
FTN
LDR
*PRI, MAIN, REF
*PRI, SUB1, REF
*PRI, SUB2, LIB, MAP
COMINP  TTY
```

The command COMINP DPROG causes the user's programs to be compiled, loaded, and the load map to be output. The file DPROG serves as documentation of the source files that make up the program and loading procedure for the program. The user would probably examine the load map at this point and save his core image. It is somewhat difficult to enter a blank line in with the editor. To do so, type a line to the editor in input mode, then go to the edit mode and give the RETYPE command with no argument. This will replace the line with a blank line.

3.3.7 COMOUTPUT: Switching Command Output from the Console to a File

Model:	COMOUTPUT *file*
Examples:	COMOUTPUT LPR
	COMOUTPUT BFO

COMOUTPUT allows you to divert system responses to commands to some file. COMOUTPUT connects *file* to unit 13. The system's responses to commands are then written on unit 13. System responses continue to be written on unit 13 until an end-of-file on COMINPUT (3.3.6) or an error is encountered. At that time system responses return to the console.

COMOUTPUT is currently not implemented.

3.4 LOAD-MODULE MANAGEMENT COMMANDS

DOS allows you to maintain files which consist of load modules. A load module is an executable program which has been generated by the loader from one or more object programs. The loader converts an object program into a load module by assigning an actual core address to each instruction in the program. Thus, in an object program, a branch to another location in the same program is represented relative to the beginning of the program or one of its storage areas, and a call to an external subroutine is represented as a flag; but in a load module, all of these references are represented as core addresses. A load module, in effect, is a core image of a program which is to be executed along with all the routines which it uses during its execution.

DOS provides several commands which you can use to handle load-module files. SAVE (3.4.1) creates a file from a core image, assigns it a name which you specify, and enters that name in your directory. RESTORE (3.4.2) reads a saved file into core. START (3.4.3) initiates execution of the core-resident load module. RESUME (3.4.4) both reads a saved file into core and initiates its execution.

When a program is in execution, the contents of two program status words and sixteen registers reflect the state of the program at every instant. The first program status word always indicates the next instruction to be executed. The second program status word indicates various conditions which occurred earlier in execution, such as arithmetic overflows and the result of the most recent comparison. The registers are special locations which various instructions use as operands. Both the program status words and the registers change from moment to moment as instructions are executed, and they always reflect the current state of a running program. Thus, if you can specify the contents of the program status words and registers in conjunction with the program's core image, you can recreate a program at any moment in its execution.

When DOS saves a load module as a file, it saves the current contents of the program status words and registers. When DOS initializes execution of a load-module file, it resets the program status words and registers to those contents. This means that you can use DOS to save a program at any moment in its execution and, at a later time, use DOS to restart execution from that point.

You will typically want to save load modules before executing them and then execute them from the beginning. In this case, the contents of all registers and the second program status word are not relevant because your program will initialize them. The first program status word is very important, however, because it indicates the next instruction to be executed - in this case, the entry point of the program. You must set this value either when you SAVE the load module or when you START or RESUME it. In general, you will want to specify the entry point when you save the load module; if, for some reason, you want to start execution at another location, you can override the saved value at that time.

3.4.1 SAVE: Saving a Load Module

Model: SAVE *file starting-address ending-address [entry-point]*...

Examples:

```
SAVE RJC 6400 9800 6450
SAVE FORTEST 6400 9AAA 6450 ( ) ( ) 0A
SAVE PRP 6400 7400
```

SAVE creates a *file* from the contents of a defined portion of core, starting at the core location of the *starting-address*, ending with the core location of the *ending-address*, and using the *entry-point*. *File* is entered in the directory to which you are attached. You can determine values for the *starting* and *ending-addresses* and *entry-point* from a loader map (3.2.6).

When a program is loaded, LDR creates a map showing core locations in the following format.

```
DEF 00000002    LIST
DEF     6400    TML$
      .
      .
      .
DEF 0000A000    MAIN
      .
      .
      .
FUL=A500
```

Arguments for the SAVE command are obtained as follows. Unless you specify an explicit load address the *starting-address* will be 6400. Addresses defined on the map for locations below 6400 are typically not part of your load module and should be ignored in specifying addresses. Thus in the example above, 6400 is the *starting-address*.

The *ending-address* is obtained by finding the first unused location in core; LDR identifies this location by printing FUL=*ending-address* at the end of the load map. In the example above, A500 is the *ending-address*. The *entry-point* is the location printed beside the name of the program being loaded. In the example shown above, MAIN is the main program and the *entry-point*.

If you wish to save a program that has been running for some time, DOS will automatically save the current value of the program-status words and registers along with the file, but you can override any of these values by specifying other entry points. If you plan to execute the saved file from the beginning, you will only want to specify *psw1*, an entry point for the program; *psw2* and 16 registers are typically not relevant during SAVE.

You can specify values for up to sixteen registers. To request default values for some registers and specify others, type a set of empty parentheses for each register for which you want a default value, as in the second example above. This convention is used only as a place-holding technique. Thus, in the second example, if you want to specify a value of 0A for register 0 rather than for register 2, you would type:

```
SAVE FORTEST 6400 9625 6450 0A
```

3.4.2 RESTORE: Reading a File into Core

Model:	RESTORE [(*directory password*)] *file*
Examples:	RESTORE (TSTONE) FORTEST
	RESTORE PTR

RESTORE reads *file* into core storage; it does not intitiate its execution. If you specify an alternate *directory* and the *passwords* match (currently blank), DOS searches for the file in that directory rather than in the current directory. If you omit a directory, DOS assumes that *file* is entered in the current directory.

RESTORE allows you to bring a program into core and patch it before executing it.

3.4.3 START: Executing an In-Core File

Model: START [*psw1*][*psw2*][r_n]...

Examples:

```
START 6400
START ( ) ( ) ( ) 61F7
START
```

START initiates execution of the program which is in core. It uses values for the program status words and registers which were SAVEd with the file. You can specify other values for the program status words and registers in the START command and thus override any saved values. You would do so if you wanted to begin execution of a program at a different point from the one at which it was saved. Type a set of empty parentheses as a place-holder for each register or program status word in the string which you want to assume the saved value.

This program might have just been RESTOREd, in which case STARTing it will cause it to begin executing, or it might have just called EXIT, in which case STARTing it will cause it to continue from the point at which it suspended itself.

3.4.4 RESUME: Restoring and Starting a File

Model: RESUME [(*directory password*)] *file* [*psw1*]

Examples:

```
RESUME RJC
RESUME (TSTONE)FORTEST 6450
RESUME MT0 7000
```

RESUME is the equivalent of RESTORE (3.4.2) and START (3.4.3) except that *psw2* and the registers cannot be specified. It reads *file* into core and initiates execution at the saved entry point or *psw1*. In all cases, DOS will initialize program status word 2 and registers with saved values. If a hexadecimal number immediately follows *file*, it is interpreted as an entry point [*psw1*]. Any other information in the line (up to a comma), represents input to the program being RESUMEd, for example, option.

DOS will assume that the file is entered in the current directory; to RESUME a file in another directory, specify *directory* and *password*.

3.4.5 PM: Displaying Program Status Words and Registers

MODEL: PM

PM types out the contents of the program status words and registers on the console. Each value is typed as an eight-digit hexadecimal number in the following format:

PSW1=*psw1*   PSW2=*psw2*
R0-7=
r_0 r_1 r_2 r_3 r_4 r_5 r_6 r_7
R8-F=
r_8 r_9 r_A r_B r_C r_D r_E r_F

3.5 USER DIRECTORY-HANDLING COMMANDS

As a DOS user, you will have your own directory in which your files are listed. When you LOGIN, you are basically "attaching" to a directory - typically your own - which becomes the CURRENT UFD. You can then process the files in that directory, which is referred to as the current directory. At any point, you can become attached to another directory - which will automatically become the current directory - and thereby gain access to the files in that directory.

3.5.1 LOGIN: Getting Started

Model:	LOGIN *directory password* [*drive*]
Example:	LOGIN MLS

LOGIN and ATTACH are identical commands. See ATTACH for a description of this command.

3.5.2 LOGOUT: Getting Off the System

Model:	LOGOUT

LOGOUT closes any open files, units, and devices and detaches you from the current directory. Every session at the console should end with LOGOUT.

3.5.3 STARTUP: Associating Disk Drives with DOS

Model:	STARTUP $drive_1$ [$drive_2$]
Example:	STARTUP 1 0

The STARTUP command has been discussed in detail in section 1.2.3 Multiple Disk Organization. STARTUP tells the system how many drives are in use and specifies a default order of searching MFD's in attempting to ATTACH to a directory. If one drive is used, it must be drive 0 with the command STARTUP 0. If the system has just been loaded, and one disk is to be used, the STARTUP command need not be given.

The number of parameters (one or two) determines the number of drives in use; and the order in which drive numbers are supplied (0 1) or (1 0) determines the order of search for the directory in a subsequent ATTACH command. The sequence specified in STARTUP can be overridden by ATTACH.

If DOS has been used after initial loading, the STARTUP command must be preceded by LOGOUT.

3.5.4 ATTACH: Gaining Access to a Directory

Model:	ATTACH *directory password* [*drive*]
Examples:	ATTACH UDIN 1
	ATTACH MED

ATTACH gives you access to the files in the specified *directory* if the *passwords* match (currently blank). This directory becomes the current directory, and subsequent commands which specify a filename but not a directory are assumed to refer to the current directory. You can access files in another directory on a temporary basis by specifying another directory in a command. The only way to change the current directory is to issue another ATTACH command.

In searching for the directory named in an ATTACH command, DOS uses the order specified in STARTUP (3.5.3). Suppose you issue the following two commands:

```
STARTUP 1 0
ATTACH UTIL
```

DOS will search for the directory named UTIL on drive 1. If the system cannot find UTIL on drive 1 it will search on the second drive named by STARTUP - drive 0.

You can override the sequence specified in STARTUP by including the drive number in the ATTACH command. If the user types

```
ATTACH UTIL 0
```

and the command STARTUP 1 0 was the last startup command typed, DOS will attempt to ATTACH to UTIL on drive 0. If a drive is explicitly identified in an ATTACH, and the directory cannot be found on that drive, DOS does not search any other drives, and an error message is printed.

3.5.5 LISTFILE: Listing the Files in a Directory

Model: LISTFILE

LISTFILE types out the names of all the files which are entered in the current directory.

3.5.6 DELETE: Deleting a File from a Directory

Model:	DELETE *file* ...
Examples:	DELETE B
	DELETE B BFOR L FTB

DELETE removes entries for the *files* from the current directory and releases the storage that the *files* occupy.

3.6 UTILITY PROGRAMS

The following external commands provide a variety of useful functions for the user.

3.6.1 LISTU: Listing the Source Files in a Directory

Model:	LISTU [*prefix*]
Examples:	LISTU XXX
	LISTU

LISTU creates two listing files in the current directory. The first file contains all source files in the directory and the second file contains a table of contents for the files listed in the first file. These two files are given names which are entered in the directory. The names are of the form *p*LIST and *p*TABL, where *p* is a 1-4 character *prefix* which you can specify. If you omit a *prefix*, the first four characters of the directory name are used (if the name is less than four characters, the entire name is used).

To obtain printed copies of the two files, you must use MEDIA. Thus, the first example above would create two listing files in the current directory named XXXLIST and XXXTABL; XXXLIST would contain all source files in the directory and XXXTABL would contain a table of contents for the files in XXXLIST. Similarly, if the current directory were TSTONE, the second example would create two files named TSTOLIST and TSTOTABL.

3.6.2 CNAME: Changing the Name of a File

Model:	CNAME *directory oldfile newfile*
Example:	CNAME MLS B1 B2

CNAME changes the name of a specified file in the directory. CNAME will locate *oldfile* is the appropriate directory and change it to *newfile*, without altering the contents of the file in any way.

3.6.3 MOVEF: Moving a File from One Directory to Another

Model:	MOVEF *file directory1 directory2*
Example:	MOVEF FILEX TSTONE TSTONX

MOVEF will insert an entry for *file* in *directory2* and delete its entry in the *directory1*. The contents of the file will not be moved. Both *directory1* and *directory2* must be on the same disk. Caution should be exercised if two disks are mounted.

3.6.4 MOVEFS: Moving Files from One Directory to Another

Model: MOVEFS *directory*$_1$ *directory*$_2$

Examples:

```
OK, MOVEFS WYMAN UTILITY
> S:PROG1
> B:PROG1
> PROG1
>
OK,
```

MOVEFS performs the same function for several files as MOVEF performs for a single file. The precautions mentioned in the description of MOVEF should be observed. The operation is terminated by a carriage return.

3.6.5 COPYFS: Copying Files

Model: COPYFS *directory*$_1$ [*drive*] *directory*$_2$ [NUM]

Example:

```
OK, COPYFS WYMAN 1 UTILITY
> S:PROG1
> B:PROG1
> PROG1
>
OK,
```

COPYFS copies one or more files from *directory*$_1$ on the specified *drive* to *directory*$_2$ on the system disk (drive zero). The original files are not deleted. Source, binary and save files may be copied. If the option NUM is specified on the command line, for each source file copied, sequence numbers are placed in positions 73-80 of each line in the file. Sequence numbering starts with 100 and is increased by an increment of 100. Sequence number format is compatible with the requirements of the batch SUPDATE program (3.2.9).

In the example given above, three files are copied from the directory WYMAN on drive 1 to the directory UTILITY on drive 0. The operation is terminated by a carriage return.

3.6.6 CARDFS: Creating a File From a Card Deck

Model: CARDFS

Example:

```
OK, CARDFS
> S:FILE1
> N:PROG2
OK,
```

Before issuing the CARDFS command, one or more card decks separated by single $EOF cards must be placed in the card reader followed by two $EOF cards, and the card reader must be readied. In response to each ">" type the file name to be assigned to the file created for each deck. All files are entered in the currently attached directory. Existing files with the same name are overwritten. The operation is terminated by the two successive $EOF cards.

3.6.7 PRINTFS: Listing Source Files on the Line Printer

Model: PRINTFS

Example:

```
OK, PRINTFS
> S:PROG1
> S:PROG2
>
```

One or more source files in the currently attached directory may be listed on the line printer. Line length must be 80 characters or less. The filename, a copy of the first line, the current date and a page number are supplied as page headings. The operation is terminated by a carriage return.

3.6.8 DMPSAV: Listing a Save File in Dump Format

Model: DMPSAV

Example: OK, DMPSAV

TITLE$_1$ = (WYMAN) PROG$_1$

MORE ? (Y OR N) N

OK,

The specified save files (S) are printed on the line printer in hexadecimal and character format. Each line is labelled at the left by a hexadecimal core address. If a directory is not specified, the file is retrieved from the currently attached directory. The user's response to TITLE$_1$ = (including an optional comment following the filename) is used as a page heading. Page numbers are supplied. If MORE is answered Y, the program requests a new TITLE$_1$ = . N terminates the operation.

3.6.9 CPRSAV: Comparing Two Save Files in Dump Format

Model: CPRSAV

Example: OK, CPRSAV

TITLE$_1$ = MEDIA (NEW VERSION)

TITLE$_2$ = (COMDIR) MEDIA

OK,

The two save files are compared word by word. Whenever a pair of words are not identical, a line is printed on the line printer, exhibiting the differing contents of the two files side by side. Blanks are inserted or print lines omitted if an exact match is formed. The print format is similar to DMPSAV format. CPRSAV is useful in detecting patches which have been applied to save files.

CHAPTER 4
SYSTEM ROUTINES AND UTILITY FUNCTIONS

4.1 INTRODUCTION

The DOS-32 MAC assembler and FTN compiler are special versions of the Honeywell-supplied OS1 assembler and compiler. The original OS1 components have been modified in order to integrate them with DOS's interactive supervisor and file system. As a result, the assembler and compiler can be invoked from the console with user commands; and the source programs which they process and the object programs which they create are files. Furthermore, FORTRAN and MAC programs themselves operate on files. Thus, all FORTRAN and MAC programs which you write use DOS routines to perform their input and output operations. These routines are described in 4.3.

DOS teletype routines available to the user are described in section 4.4. Section 4.5 describes utility routines which are the library (file LIBRARY in directory LIBDIR). Two of these routines (EXIT and SSWS) are special DOS utility routines instead of library routines. Section 4.6 describes peripheral device routines in the library. Section 4.7 describes how to use FORTRAN Input-Output statements. Section 4.8 describes overlay management routines.

Because DOS is a FORTRAN-oriented system, the models and examples in this chapter reflect FORTRAN use. In the models, all words in CAPITAL letters and all punctuation marks must be coded as shown; all words in *lower-case italics* are arguments, which you must supply as required. Keypunching and coding must follow standard Honeywell 632 conventions, as presented in the *Series 32 FORTRAN*

Manual (Publication M-423; June, 1969). If you are programming in MAC, follow standard MAC linkage conventions for accessing subroutines, as described in the *MAC-32 Assembler Manual* (Publication M-421; November, 1968). Routines which return values as variables in FORTRAN return them in register 13 in MAC.

4.2 FORTRAN INTEGERS

FORTRAN stores integer data in bits 0-30 of each word and sets bit 31 to zero. This means that if you examine the actual bit configuration of a word, you will find a number with a value which is twice that of the stored integer. You will have to make appropriate mental adjustments - by right-shifting the word or dividing it by two - whenever you try to deal with the machine-language equivalent of the program (e.g., debugging the program using a dump, or communicating with a MAC program). It should be emphasized, however, that this convention will normally not affect you in any noticeable way.

Many of the DOS character-manipulation utility routines - e.g., LOC (4.5.30), SLH (4.5.9), LCHAR (4.5.14) - expect to find data in integer position or to store values in integer position. If you want to use these routines on values that are not stored in integer position, you will have to compensate for the storage convention by performing the appropriate shift operation. For those routines where confusion may occur, *int* is used for variables in integer position and *word* is used for variables in normal position. *Array* normally refers to an array of *words*. See appendix I for a description of other nmenonic names.

4.3 FILE-SYSTEM ROUTINES

All I/O for programs which are run under DOS is handled by the DOS file system, which controls all access to files. FTN and MAC have been modified to process source-program files and to generate object-program files. In addition, run-time routines have been modified to use file-system routines for I/O. In FORTRAN, this is an added capability, and you have the choice of using FORTRAN Input-Output statements (4.7) or DOS routines. In MAC, this is a requirement, and you must use DOS routines instead of the OS1 routine URC.

The file system automatically maintains information about the characteristics of all files and keeps track of where they are stored. When you want to process a file, you assign it to a unit (3.3.4). Associated with each unit is a control block. When a file is assigned to a particular unit, that unit's control block contains descriptive information about the file. At that time, a buffer is allocated to the unit. During processing of the file, the assigned buffer will contain a record which has just been read from the file or is about to be written onto the file, as appropriate.

You can call five file-system routines from your programs to access and process files. OPEN (4.3.1) associates a file with a DOS unit and prepares the file for reading or writing, as specified. CLOSE (4.3.2) terminates operations on a file. BRWFIL (4.3.3) performs actual reading and writing of records. DELETE (4.3.4) returns all records in a file to the availability pool and removes the file entry from the appropriate directory. REWIND (4.3.5) positions a file at its beginning. OPEN, CLOSE and DELETE can all be performed from the console and it may simplify your program if you take advantage of this. However, a program which does its own OPENs and CLOSEs is simpler to operate. If you wish to OPEN your own files, use GTNAMS (4.5.29).

ATTACH (4.3.6) allows a user to change directories under program control. COMANL (4.3.7) allows a user to retrieve items from the DOS command line. This routine is useful in generating new DOS commands. If the user specifies an alternate return on file system routines, control will be passed to the alternate return on an error condition. In the case of OPEN and BRWFIL, more than one type error can cause control to go to the alternate return. The user calls GETERR (4.3.8) to retrieve a block of information about the error. After examining the error, he may wish to print the error message that would have been printed if alternate return was zero, then return to DOS command level. PRTERR (4.3.9) is called for this purpose.

4.3.1 OPEN: Opening a File

Model: CALL OPEN (*key*, *file*, *unit*, *altrtn*, *newfil*, *directory*, *password*)

Examples:

```
CALL OPEN (1,'LISTN', 3, STMNT, 0, 0, 0)
CALL OPEN (2, TAB, 6, 0, NFIL, TSTONE, PSWRD)
```

OPEN associates *file* with *unit* and sets up a control block which contains all relevant information about the open file. The *unit* must be an integer 1-7. The *key* is an integer 1-3 which specifies the type of operation for which the file is being opened:

Argument	*Meaning*
key	1=reading 2=writing 3=reading and writing

Argument	*Meaning*
file	The name of the file being opened.
unit	The number (1-7) of the unit through which the file is to be read or written.
	(When specifying the name itself in this call, put it in single quotes and make sure that the literal is 8 characters long, including blanks)

Other arguments are specified as follows:

Argument	*Meaning*
altrtn	Data name which has been assigned (in an ASSIGN statement) to a statement number in your FORTRAN program. In case of error, control will pass to the specified statement. If *altrtn* is zero (as in the second example above), an error condition will cause execution of the program to terminate, an error message will be typed out on the console, and control will return to the DOS supervisor.
newfil	Array of one or three words in length, used only when the file is being opened for output. If the file that you are creating is a program or data file, *newfil* must be a one-word array which contains a zero. If the file that you are creating is a directory, *newfil* must be a three-word array: the first word must contain a two, and words 2-3 must contain a password. If you are opening an input file, *newfil* must be zero.
directory	The name of the directory in which an input file is entered, if not the current directory. If you are using the current directory or are opening a file for output, *directory* must be zero or blank.
password	Two-word array which contains the password for *directory*, if *directory* is nonzero. If *directory* is zero, *password* must be zero.

If *altrtn* is not zero, ERRVEC, a DOS array is set as follows:

ERRVEC(1)=OPEN

ERRVEC(2)=blanks

ERRVEC(3)=1	if unit already open
=2	if no ufd attached
=3	if name not in ufd

4.3.2 CLOSE: Closing a File

Model:	CALL CLOSE (*file-or-unit*)
Examples:	CALL CLOSE (1)
	CALL CLOSE (FILUN)
	CALL CLOSE ('ALL')

CLOSE deactivates an open unit or file. If *file-or-unit* is an integer 1-7, it is interpreted as a unit, and CLOSE closes the specified unit; if *file-or-unit* is a 1-8 character string (beginning alphabetic), it is interpreted as a filename, and CLOSE closes the specified file. If the unit or the file is not open, no action is taken. CLOSE ('ALL') closes all open units and files.

If you want to close a file which is not in the current directory (i.e., you specified a *directory* when you OPENed it), you must close it by unit number, not by name.

4.3.3 BRWFIL: Reading, Writing, or Manipulating a File

Model:	CALL BRWFIL (*key, unit, array, n, rel, altrtn*)
Examples:	CALL BRWFIL ('RFWD', 3, BUFF, FSIZ-7, 10, RTN)
	CALL BRWFIL (4HWFLN, UNIT, UFDNT(8), 120, 0, 0)
	CALL BRWFIL ('CFRW', 1, 0, 0, 0, ERR)

BRWFIL reads, writes, rewinds, or truncates the file which has been opened on *unit* (an integer 1-7). The operation to be performed is determined by *key*, a four-character code which is typically represented as a literal. Specify *key* as follows:

Character(s)	*Value*	*Meaning*
First	R	Read
	W	Write
	C	File control

Character(s)	*Value*	*Meaning*
Second	F	Forward
	B	Backward (currently not implemented)
Third-Fourth	WD	Word mode
	LN	Line mode
	TR	Truncate
	RW	Rewind

Reading and writing can be in word mode or line mode. In word mode, data are stored as bit strings. In line mode, data are stored in a condensed format. Whereas any type of data can be transcribed in word mode, only character data can be transcribed in line mode. Once a file has been written in line mode, it must be read in line mode. Similarly, once a file has been written in word mode, it must be read in word mode.

The read operations which are currently supported are RFLN and RFWD; the write operations are WFLN and WFWD. The following abbreviations are acceptable: 'W'='WFWD', 'R'='RFWD', 'TRUN'='CFTR', and 'RWND'='CFRW'. If BRWFIL is operating in word mode, exactly *n* words will be read into or written from *array*. If BRWFIL is operating in line mode, one line will be read or written. In a write operation, line length is determined by *n*. The *array* must not contain newline (linefeed) characters. In read operation, the length *n* of the *array* to be filled must be large enough to receive the line as written. A short line will be padded with blank characters to fill *array*. A line of length greater than *array* will result in an error condition.

A file can be repositioned before reading or writing if *rel* is nonzero. The file will be moved forward *rel* words if *rel* is a positive integer, or backward *rel* words if *rel* is a negative integer. Because relocation is by words, *rel* should be zero if BRWFIL is operating in line mode.

In DOS, you may change the contents of individual words after you have created them. You do so by writing in the file again, possibly after relocating. To make this capability general and flexible, DOS does not write a new end-of-file when you finish writing, unless you have written beyond the old end-of-file. If you want a new end-of-file, you must use the truncate operation.

In a truncate operation, the value of *key* must be CFTR; in a rewind operation, the value of *key* must be CFRW. Truncating a file is terminating it, by putting an end-of-file after the last data which were in *array*. Rewinding a file is repositioning it at its beginning. When BRWFIL is performing a file-control operation, the values of *array*, *n*, and *rel* should be zero.

If *altrtn* is nonzero, control will pass to the specified statement number in your FORTRAN program when an end-of-file (on input) or error condition occurs. Specify *altrtn* as a data name which has been assigned an integer value in an ASSIGN statement. If *altrtn* is zero, an error will cause control to return to the system.

A description of the error codes returned by BRWFIL through ERRVEC at abnormal termination is given below.

ERRVEC(1)	=	BRWF
ERRVEC(2)	=	IL
ERRVEC(3)	=	2 - illegal key code
	=	4 - status error
	=	5 - line longer than array in RFLN call
	=	6 - relocate error

ERRVEC(3)	=	7 - *rel* not equal to 0
	=	8 - control type error
	=	10 - data type error
	=	12 - end of file

In the case of end of file

ERRVEC(4)	=	unit
ERRVEC(5)	=	number of words not read
ERRVEC(6)	=	number of bits in last word.

4.3.4 DELETE: Deleting a File

Model: CALL DELETE (*file, altrtn*)

Examples: CALL DELETE (BFOR1, RTN1)

DELETE releases all storage which was used by *file* and removes the file's entry from the current directory. You cannot delete a file in another directory. If *altrtn* is nonzero and *file* is not found in the directory, control will pass to the specified statement number in your FORTRAN program. If *altrtn* is zero, or if any other type of error occurs, control will return to the system.

4.3.5 REWIND: Repositioning a Unit at the Beginning of a File

Model: CALL REWIND (*unit, altrtn*)

Examples: CALL REWIND (5, 0)

CALL REWIND (1, NOPN)

REWIND repositions the file which is connected to *unit* (an integer 1-7) at its beginning. If the unit is not open and *altrtn* is nonzero, control passes to the specified statement number in your FORTRAN program. If *altrtn* is zero and an error occurs, control will pass to the system.

4.3.6 ATTACH: Gaining Access to a Directory

Model: CALL ATTACH (*directory, drive, password,* .TRUE. *altrtn*)

ATTACH searches the MFD on disk *drive* for *directory*. If *drive* is -1, all active disks are searched in sequence for *directory*. If *directory* is found and *password* matches the password associated with *directory*, the current file directory is set to *directory*. All passwords are currently blank, so *password* should be a two-word array containing 8 spaces. Control is sent to *altrtn* if nonzero and *directory* is not found in the MFD.

4.3.7 COMANL: Performing Lexical Analysis of User Commands

Model:	*rtnkey* = COMANL (*key, array*)
Example:	RTNKEY = COMANL (1, VEC2)

COMANL performs lexical analysis of user commands and looks ahead to the next item in the command line. The value of *key* specifies the item type. If the next item is of that type and *key* $\neq$ 0, *rtnkey* is set to *key* and the item is put in *array*, a one or two-word *array*. If the next item is not of the type specified by *key*, *rtnkey* is set to zero and nothing is put in *array*.

The *key* argument can take the following values:

0	next item can be of any type.
1	next item should be a command name (alphanumeric string beginning with an alphabetic character and followed by a blank); lower-case letters are changed to capital letters.

2 next item should be a name (alphanumeric string beginning with an alphabetic character); lower-case letters are not changed to capital letters.

3 next item should be a left parenthesis (indicates introduction of an option).

4 next item should be a right parenthesis (indicates conclusion of an option).

5 next item should be a parameter (hexadecimal or octal number).

6 next item should be a terminator (comma or end-of-buffer, which is transmitted as a new-line character).

If *key* is 6 and a terminator is found, the pointer is not moved beyond the terminator, and subsequent calls will still show a terminator as the next item in the command line. If *key* is 0, *rtnkey* is set to the number which corresponds to whatever item type is found. In this case, an alphanumeric string beginning with an alphabetic character is treated as a name rather than as a command name, i.e., *rtnkey* is set to 2, not 1.

The following program is an example of the use of COMANL

```
      INTEGER NAME(2)
      IKEY = COMANL(2,NAME(1))
      IF(IKEY EQ. 0) GO TO 10
      WRITE(100, 1) NAME(1), NAME(2)
      CALL EXIT
C
   10 WRITE(100, 2)
      CALL EXIT
    1 FORMAT(2A4)
```

```
2     FORMAT('IMPROPER NAME')
      END
```

Assume the user has loaded and saved the program as *PROG. The user types

```
RESUME *PROG XXX
```

The program calls COMANL to pick up the next item. DOS has already called COMANL to process the RESUME and *PROG items. The next item picked up will be XXX which is a name, so IKEY is set to 2 and the program types XXX and returns to DOS on the call to EXIT

4.3.8 <u>GETERR: Storing Error Information</u>

Model:	CALL GETERR(*array*)
Example:	CALL GETERR(VECLOC)

If the user calls subroutine OPEN or BRWFIL with an alternate return of nonzero, then control will be passed to the alternate return in case of an error. More than one error causes control to pass to alternate return in the case of these two subroutines. To determine the type of error and other information about the error, the user calls GETERR which transmits the 16 word array ERRVEC from DOS COMMON to *array*. Both subroutines OPEN and BRWFIL set ERRVEC before passing control to alternate return.

4.3.9 <u>PRTERR: Typing an Error Message</u>

Model: CALL PRTERR

If the user specifies a nonzero alternate return for OPEN or BRWFIL, then an error will cause control to pass to alternate return. The user determines the type of error encountered by calling GETERR (4.3.8). If the user decides not to continue processing after a certain error type has been received, he may have DOS print out the error message that would have been typed if alternate return were zero, then return to DOS command level by calling PRTERR.

4.4 TELETYPE ROUTINES

DOS allows you to access its teletype I/O routines which control reading and writing of characters and lines. This facility enables you to write interactive programs, which can have generalized capabilities and depend on the user's response to runtime questions for the specific parameters of a job. You may also access the teletype from FORTRAN.

4.4.1 RDASR: Reading a Line from the Teletype

```
Model:          CALL RDASR (array, n)
Examples:       CALL RDASR (LST1, 20)
                CALL RDASR (TITLE(8), 5)
```

RDASR reads one line from the teletype and places it in *array*. The length of the line must be no greater than *n* words; the length of *array* must be at least *n* words. The end of a line of input is signalled by a carriage return or line feed. There will be 4 characters in each word. If too few characters typed, then blanks will be inserted.

4.4.2 WRASR: Typing a Line on the Teletype

```
Model:          CALL WRASR (array, n)
Examples:       CALL WRASR (OUTP, 20)
                CALL WRASR (LINE(10), 35)
```

WRASR types out onto the teletype one line of output as stored in *array*. The length of this line is *n* words, where *n* is an integer 1-20; if *n*=0, 20 is used.

4.4.3 T1IN: Reading a Character from the Teletype

Model:	CALL T1IN (*int*)
Examples:	CALL T1IN (X)
	CALL T1IN (ALPHA)

T1IN reads a signal character from the teletype and places it in integer position in *int*, a one-character data variable. Null characters are ignored, and control is returned when a valid character has been read. Carriage returns and line feeds both set the value of *int* to indicate a line feed ($0A).

4.4.4 T1OU: Typing a Character on the Teletype

Model:	CALL T1OU (*int*)
Examples:	CALL T1OU (A)

T1OU types a single character *int* on the teletype. The character must be specified as a data variable in integer position; it cannot be a literal. A line feed will be preceded by a carriage return, but a carriage return will not be preceded by a line feed.

4.4.5 TNOU and TNOUA: Typing a Character String on the Teletype

Models:	CALL TNOU (*array*, *n*)
	CALL TNOUA (*array*, *n*)
Examples:	CALL TNOU (ERMS, 28)
	CALL TNOUA ('TYPE Y TO CONTINUE', 18)

TNOU and TNOUA type *n* characters from *array*. Characters must be packed in the array four per word, but *n* does not have to be an even multiple of four. *Array* may be a literal or may have been set up by some other means. Exactly *n* characters will be typed, and the last word, if incomplete, need not be padded with null or blank characters.

TNOU provides a carriage return/line feed after typing, but TNOUA does not. Both routines precede an embedded line feed with a carriage return but do not precede a carriage return with a line feed.

4.4.6 TOHEX: Converting a Word to Hexadecimal and Typing It

Model: CALL TOHEX (*word*)

Example: CALL TOHEX (IRDC)

TOHEX converts the *word* into an eight-digit hexadecimal character string and types it. Leading zeroes are typed as zeroes.

4.5 UTILITY FUNCTIONS

DOS allows you to access a set of utility routines which provides additional logical, bit- and character-manipulation and computation capabilities. All of these routines are stored in LIBRARY, a file which is listed in the system file directory LIBDIR. If you use any of these routines in your program, you must specify the LIB option when loading (3.2.6).

4.5.1 AND: Forming Logical Product

Model: *wordc* = AND (*worda*, *wordb*)

Example: BIN = AND (LIST, M1)

AND returns as an integer value the logical product of *worda* and *wordb*. Each bit in X is set to a one if the corresponding bits in *worda* and *wordb* are both ones. If either is zero, the bit is set to zero.

4.5.2 OR: Forming Logical Sum

Model: *wordc* = OR (*worda*, *wordb*)

Example: T = OR (CNT, '$000AE200')

OR returns as an integer value the logical sum of *worda* and *wordb*. Each bit in X is set to a one if either of the corresponding bits in *worda* or *wordb* is a one. If both are zeroes, the bit is set to zero.

4.5.3 XOR: Forming Exclusive OR

Model: *wordc* = XOR (*worda*, *wordb*)

Example: X = XOR (AL1, SCN)

XOR returns as an integer value the exclusive OR of *worda* and *wordb*. It sets each bit in X to a one if either but not both of the corresponding bits in *worda* and *wordb* is a one. If both are zeroes or ones, the bit is set to zero.

4.5.4 COMPL: Forming Complement

Model: *wordb* = COMPL (*worda*)

Example: C2 = COMPL (C1)

COMPL returns as an integer value the complement of *worda*. It sets each bit in X to a zero if the corresponding bit in *worda* is a one, and each bit to a one if the corresponding bit is a zero.

4.5.5 LH: Retrieving Left-Half of Word

Model: *int* = LH (*word*)

Example: J1 = LH (MS1)

LH returns the left-half of *word* as an integer value in integer position.

4.5.6 RH: Retrieving Right-Half of Word

Model: *int* = RH (*word*)

Example: INUN = RH (NEXT)

RH returns the right-half of *word* as an integer value in integer position.

4.5.7 LHC: Retrieving Left-Half of Word, Indirect

Model: *int* = LHC (*pointer*)

Example: IC2 = LHC (E)

LHC returns the left-half of the word that has its address in *pointer* as an integer value in integer position.

4.5.8 RHC: Retrieving Right-Half of Word, Indirect

Model: *int* = RHC (*pointer*)

Example: ISRW = RHC (MSK2)

RHC returns the right-half of the word that has its address in *pointer* as an integer value in integer position.

4.5.9 SLH: Storing Number in Left-Half of Word

Model: CALL SLH (*word*, *int*)

Example: CALL SLH (N, SUM)

SLH stores the number in integer position of word *int* in the left-half of *word*.

4.5.10 SRH: Storing Number in Right-Half of Word

Model: CALL SRH (*word*, *int*)

Example: CALL SRH (NR, SUM)

SRH stores the number in integer position of word *int* in the right-half of *word*.

4.5.11 MAKWRD: Forming a Word from Two Halfwords

Model: *word* = MAKWRD (*inta*, *intb*)

Example: MSK = MAKWRD (IPT1, IPT2)

MAKWRD returns as an integer value a word which contains the value in integer position of word *inta* in its left-half and the value in integer position of word *intb* in its right-half.

4.5.12 LHWRD: Retrieving a Halfword

Model: *int* = LHWRD (*array*, *n*)

Example: IPAR = LHWRD (SCAN, 4)

LHWRD returns the *n*th halfword of *array* as an integer value in integer position.

4.5.13 SHWRD: Storing a Halfword

Model: CALL SHWRD (*array*, *n*, *int*)

Example: CALL SHWRD (PLIN, 15, ICNT)

SHWRD stores in the *n*th halfword of *array* the value in integer position of word *int*.

4.5.14 LCHAR: Retrieving a Character

Model: *int* = LCHAR (*array*, *n*)

Example: K2 = LCHAR (OPTNS, 5)

LCHAR returns the *n*th character of *array* as an integer value in integer position.

4.5.15 SCHAR: Storing a Character

Model:	CALL SCHAR *(array, n, int)*
Example:	CALL SCHAR (ERMSG(6), 7, ICODE)

SCHAR stores in the *n*th character position of *array* the character in integer position of word *int*. *Int* may not be a literal.

4.5.16 PUTC: Inserting a Character in an Array

Model:	CALL PUTC *(int, array)*
Example:	CALL PUTC (A, CWRD)

PUTC inserts *int*, a character in integer position, in the first blank in *array*, a two word array. *Int* may not be a literal. PUTC is typically called eight times in succession, when *array* has been blanked before its first call. Calls to PUTC after the eighth are ignored.

4.5.17 LBIT: Retrieving a Bit

Model:	*int* = LBIT (*array, n*)
Example:	IPTR = LBIT (CODES, 7)

LBIT returns the *n*th bit of *array* as an integer value in integer position.

4.5.18 SBIT: Storing a Bit

Model:	CALL SBIT *(array, n, int)*	*n*=1,...32
Example:	CALL SBIT (LST(2), 3, 0)	

SBIT places a zero in the *n*th bit of *array* if *int* is zero or a one if *int* is nonzero.

4.5.19 RT: Retrieving an Integer

Model: *int* = RT *(n, word)*

Example: INT = RT (16, J)

RT returns as an integer in integer position the rightmost *n* bits of *word a*. The rightmost bit a *a* is also moved into the rightmost bit of I.

4.5.20 SSWS: Testing a Sense Switch

Model: *logical* = SSWS *(n)*

Example: SW1 = SSWS (1)

DOS maintains fifteen boolean variables to serve <u>in place of</u> the sense switches on the System Control Panel. These "pseudo sense switches" can be set and reset by the user (see section 1.2.6) and tested by an executing program.

SSWS is the logical function which returns a value of .TRUE. if pseudo switch <u>*n*</u> is set, and .FALSE. otherwise; <u>*n*</u> is any of the integers 1 through 15, which identify respectively pseudo sense switches A through O.

4.5.21 CHKSUM: Computing Checksum

Model: *int* = CHKSUM (*array*, *n*)

Example: CS1 = CHKSUM (LIN(7), 3)

CHKSUM adds together the first *n* words in *array* and returns their simple algebraic sum.

4.5.22 COMEQV: Comparing Command-Names

Model: *logical* = COMEQV *(arraya, arrayb)*

Example: T = COMEQV (KEY(1), 4HLSTN)

COMEQV compares *arraya* and *arrayb*. Both *arraya* and *arrayb* must be one- to eight-character strings which have no embedded blanks, and *arraya* can be shorter than *arrayb*. COMEQV returns a value of .TRUE. if the first *n* characters of *arraya* are identical to the first *n* characters of *arrayb*, where *n* is the number of non-blank characters in *arraya*. If not, it returns .FALSE.

4.5.23 NAMEQV: Comparing Names

Model:	*logical* = NAMEQV (*arraya*, *arrayb*)
Example:	PAR1 = NAMEQV (PARAM(6), KEY(6))

NAMEQV compares *arraya* and *arrayb*, both of which must be one- to eight-character strings. NAMEQV returns a value of .TRUE. if *arraya* and *arrayb* are identical and .FALSE. if they are not.

4.5.24 DCNVRT/HCNVRT: Converting a Number from Internal Format to Decimal or Hexadecimal

Model:	CALL DCNVRT (*int*, *array*, *n*)
	CALL HCNVRT (*word*, *array*, *n*)
Examples:	CALL DCNVRT (NUM1, OUTL(5), 4)
	CALL HCNVRT (HNST, HLST, 10)

DCNVRT/HCNVRT converts *int* or *word*, a number in internal format, to decimal or hexadecimal representation for printing or typing. The converted number is right-adjusted in *array*, an array of *n* words in length. Leading zeroes in *int* or *word* are suppressed, and, if necessary, *array* is left filled with blanks. For decimal conversion, *int* is treated as an integer in integer position; all 32 bits of *word* take part in hexadecimal conversion.

4.5.25 EXIT: Returning to the System

Model: CALL EXIT

When your program has completed and you wish to return control to DOS, CALL EXIT. EXIT allows the user to issue commands and to return via a START command. DOS will type OK.

4.5.26 GTHEX: Searching for Hexadecimal Number in Character String

Model: *word* =GTHEX(*array*,*n*,*m*)

Examples: X=GTHEX(XARR,7,19)

X=GTHEX(NRY,CNT1,CNT2)

GTHEX searches for the first hexadecimal number in a character string, starting with the *n*th character and stopping at the *m*th. If a hexadecimal number is found, GTHEX returns the number and sets *n* to the index of the character following the entry. If none is found, GTHEX returns zero and *n* is set to *m*+1. Some caution is required in testing the returned value of *n*, because the compiler sometimes optimizes in such a way that changes in arguments of functions may be ignored.

4.5.27 LHEX: Retrieving a Hexadecimal Digit

Model: *int*=LHEX (*array*, *n*)

Example: X=LHEX(STRG,19)

LHEX returns the *n*th hexadecimal digit of *array* where the index *n* starts at 1.

4.5.28 GTREGS: Retrieving the Address of the Register Save Area

Model: *pointer*=GTREGS (0)

If you are writing a FORTRAN subroutine and need to know where the registers (1 through 15) from the calling routine were saved, use GTREGS. GTREGS works only when called by a standard FORTRAN routine and should be called immediately upon entry.

4.5.29 GTNAMS: Retrieving Directory and File from a Character String

Model: CALL GTNAMS (*array*,*m*,*n*,*directory*,*password*,*file*, *flag*,*altret*)

Example: CALL GTNAMS (LINE,1,40,UFD,PSW,FNM,FOUND,NONAME)

GTNAMS searches the character string in *array* starting at character position *m* and continuing through *n*. If the form (*name*1) is encountered first, the two-word array *directory* is set to *name* 1. If the form (*name*1 *name*2) is encountered, both *directory* and *password* are filled in. The search continues until the form *name*3 is encountered or until the *n*th position is reached. If found, *name*3 is assigned to *file*, and the logical variable *flag* is set to .TRUE. Otherwise, *flag* is set to .FALSE. and arguments corresponding to missing fields are set to blanks. The exit *altret* is taken if invalid characters are found.

4.5.30 LOC: Retrieving the Address of a Variable

Model: *pointer* = LOC (*word*)

Example: IVAL = LOC (VARX)

LOC returns the machine address of word *a*.

4.5.31 LIST: Referring an Absolute Core Address

Model: *word* = LIST (*pointer*)

Example: IVAL = LIST (VARX)

LIST is an array defined to start at absolute location 1. To use LIST, the user must have in his program the statement

```
COMMON /LIST/LIST(2)
```

as the loader has built into it the definition of the LIST COMMON area to start at location 1. The statement I = LIST (100) puts the contents of location 100 into variable I. The statement LIST (200) = I puts the contents of variable I into location 200. LIST and LOC may be used as basic building blocks for creating, manipulating, and referring list structures.

4.5.32 SRHC: Storing Number in Right-Half of Word, Indirect

Model:	CALL SRHC (*pointer, int*)
Example:	CALL SRHC (N, SUM)

SRHC stores the number in integer position of word *int* in the right-half of the word whose address is in *pointer*.

4.5.33 SLHC: Storing Number in Left-Half of Word, Indirect

Model:	CALL SLHC (*pointer, int*)
Example:	CALL SRHC (N, SUM)

SLHC stores the number in integer position of word *b* in the right-half of the word whose address is in *pointer*.

4.6 PERIPHERAL DEVICE ROUTINES

DOS provides the user with several routines which are used in handling the system's various peripheral devices. Those routines currently implemented allow cards to be read (4.6.1 DOCARD), lines to be printed (4.6.2 DOPRIN), magnetic tape to be used (4.6.3 MT7IOC). The user may communicate to the line printer and card reader from FORTRAN through unit 101.

4.6.1 DOCARD: Reading a Card

Model:	CALL DOCARD (*array*)
Example:	CALL DOCARD (CDBFT)

DOCARD reads a card. If a bad status condition is discovered, the program issues an appropriate diagnostic. If the operation corrects the condition and types a carriage return, DOCARD will try again. *Array* must be 20 words long.

4.6.2 DOPRIN: Printing a Line

Model:	CALL DOPRIN (*array*)
Example:	CALL DOPRIN (PRBF)

DOPRIN prints one line. If a bad status condition is discovered, the program types status in uninterpreted form and prints the line again. *Array* must be 31 words long. The first word of *array* is used for carriage control and is not printed. Character 1 of the control word is interpreted as follows:

Character	*Effect*
1	Skip to top of page
+	Suppress spacing
0	Double-space
Other	Single-space

4.6.3 MT7IOC: Controlling Magnetic Tape

Model:	CALL MT7IOC (*key,array,n,length,mode,tcu,mtt, status*)
Example:	CALL MT7IOC (1, IRAY, 200, 20, 2, 1, 0, ISTAT)

MT7IOC provides synchronous input, output, and control of seven-track magnetic tapes. According to specified arguments, this module builds an appropriate channel program and waits for completion. It provides user-access to all seven-track magnetic-tape operations and modes. Upon completion, a status indication is returned. Control is then returned to the caller.

Arguments are interpreted as follows:

key Specifies operation:

1. Read a full record and set *length* to the number of words read. If *array* is too short, extra bytes are ignored. If end-of-file is encountered, no data are transmitted (*length*=0); if end-of-tape is encountered, the record is transmitted (*length*=0). In case of error, nine additional reads are attempted; if still in error, the data are passed as read and the tape is repositioned at the next record.
2. Write a record of *n* words. In case of error, up to nine backspace-erase-write sequences are attempted; if still in error, a backspace command and an erase command are issued (preparing the tape for another record). If end-of-tape is encountered, the record is written.
3. Write tape mark.
4. Rewind (no-op if at beginning of tape.)
5. Rewind and unload (no-op if at beginning of tape).

key	6. Backspace record.
	7. Skip record.
	8. Backspace to start of file.
	9. Skip to end of file.
buffer	An array.
n	Integer 1-4096 specifying length of *array* in words.
length	Length of record in words for read operation (set by MT7IOC).
mode	Format and parity;
	1. Word format, odd parity/
	2. Character format, even parity (BCD).
	3. Character format, odd parity (binary).
tcu	Controller address:
	1. Controller 1.
	2. Controller 2.
	3. Controller 3.
	4. Controller 4.
	(Currently only one controller, controller 1, is available.)
mtt	Transport number (indicated by dial on tape drive):
	0. Drive 0.
	1. Drive 1.
	2. Drive 2.
	3. Drive 3.
status	8-bit status set by controller and stored in integer position.
	Values are interpreted as follows:

Bit Pair	*Value*	*Meaning*
0,1	00	Controller malfunction.
	01	Drive not ready or more than one readied drive set to same transport number.
	10	Drive is write-protected, rewinding, or unload.
	11	Drive is write-permitted.
2,3	00	Reel not available because rewinding, unload, not readied, or run off physical end of tape.
	01	End-of-tape detected.
	10	Reel position at beginning.
	11	Reel ready for read or write operation (not at beginning or end of tape).
4,5	00	Parity error.
	01	Tape mark read, written, or sensed (e.g., as in skip-file operation).
	10	Record length exceeded *length* or an all-zero frame was converted when writing even parity (*mode*=2).
	11	Data are normal.
6,7	00	One byte of last word transmitted.
	01	Two bytes of last word transmitted.
	10	Three bytes of last word transmitted.
	11	Last word transmitted in full.

4.6.4 P1IN: Reading a Character from Paper Tape

Model: CALL P1IN (*int*)

Example: CALL P1IN (X)

P1IN reads a single character (*int*) from the paper-tape reader. When control is returned, the reader has been stopped. P1IN reads in coded mode--it skips null and delete characters and zeroes the high-order bit (unless the parity of all 8 bits is odd, in which case it makes the high-order bit a 1). P1IN returns a line feed

($0A) for a carriage return ($0D).

4.6.5 P1OU: Punching a Character

Model: CALL P1OU (*int*)

Example: CALL P1OU (A)

P1OU punches a single character (*int*) on the high-speed punch. When control is returned, the punch has been stopped. P1OU punches in coded mode--the seven low-order bits are punched as they are and the high-order bit is punched to make it an even-parity character.

4.6.6 PNOU: Punching a Character String

Model: CALL PNOU (*array, n*)

Example: CALL PNOU (ERMS, 28)

PNOU punches *n* characters from an *array* which is in packed format. Any unused bytes in the last word are ignored. When control is returned, the punch has been stopped. PNOU punches in coded mode, which uses the high-order bit to make the character even parity.

4.6.7 P1INB: Reading a Binary Character from Paper Tape

Model: CALL P1INB (*int*)

Example: CALL P1INB (X)

P1INB reads in one character (*int*) in binary mode from the paper-tape reader. When control is returned, the reader has been stopped.

4.6.8 P1OUB: Punching a Binary Character

Model: CALL P1OUB (*int*)

Example: CALL P1OUB (A)

P1OUB punches one character (*int*) in binary mode on the high-speed punch. When control is returned, the punch has been stopped.

4.6.9 PNOUB: Punching a Binary String

Model:	CALL PNOUB (*array*, *n*)
Example:	CALL PNOUB (ERMS, 28)

PNOUB punches *n* characters in binary mode from an *array* which is in packed format. When control is returned, the punch has been stopped.

4.7 USING FORTRAN INPUT-OUTPUT STATEMENTS UNDER DOS

Device numbers 1-7 refer to DOS units 1-7, and allow users to read or write DOS files. Users must either use I/O control commands or subroutine OPEN to connect files to units before attempting to issue READ or WRITE statements for units 1-7. After the files are read or written, they must be closed through use of subroutine CLOSE or the CLOSE command.

Device 100 issues READ and WRITE statements to the teletype. Device 101 in a READ statement will read cards from the card reader and in a WRITE statement will print lines on the line printer. The ERR and END features of the READ statement have not been implemented for the card reader. Any error will cause a message to be typed, a wait to occur for the user to prepare the card reader to reread his card, and the program to continue when a carriage return is typed. If the card reader runs out of cards, a message is typed, the reader waits for the user to add more cards, and execution is continued when carriage return is typed. The $EOF card has no special meaning and is read as any other card.

For disk I/O, FORTRAN programs can read and write two classes of records, binary and BCD. Binary records are read and written with unformatted READ and WRITE statements. FORTRAN I/O routines expect files to consist of records of one class only. The class of a particular file is established by the first READ or WRITE instruction addressed to it within a particular program. If a later attempt is made within a program to access that file with an instruction of a different class, FORTRAN error 39 will be generated.

Each BCD record is in line mode and cannot exceed 30 words. If the data list and format statements of a WRITE instruction generate more than 30 words, the data beyond the 30-word limit are lost. Binary records exist on external media as a series of 124 words, written in word mode. DOS retains the concept of a physical record. Under DOS, data in a file are maintained as a continuous series of words; but the I/O routines, when dealing with a binary file, consider each 124 words to be a physical record.

Each binary logical record (i.e., the data read or written by one binary READ or WRITE statement) consists of an integral number of 124-word physical records. The first word of each of these physical records is a control word. The first half of this control word is a count of the number of physical records within the logical record; the count of the last (or only) physical record of the current logical record is maintained in negative form. The second half of the control word is a count (≤ 123) of the number of data words in that physical record. If this count is less than 123, the remainder of that series of 124 words is garbage.

I/O control statements other than READ and WRITE may be used with DOS files. The REWIND statement may be used in place of the REWIND subroutine (4.3.5) to reposition a unit to the beginning of a file. The BACKSPACE statement may only be used with binary records as it has not been implemented for BCD records. The ENDFILE statement is not necessary as DOS needs no actual mark to indicate end of file. If a user is overwriting an existing file and wants to indicate an end-of-file condition at the current position of the file, he should call subroutine BRWFIL (4.3.3) with a truncate key.

4.8 OVERLAY-MANAGEMENT ROUTINES

DOS allows you to access routines which aid in overlay management.

4.8.1 RESTOR: Bringing a Load Module into Core

Model:	CALL RESTOR (*file*,*array*,*0*,*altrtn*)
Example:	CALL RESTOR (FILEX,USRVEC,0,ALT2)

RESTOR restores a saved *file* by placing the saved program status words and registers in *array* and the saved core image of *file* in core position *sa* (starting address) through *ea* (ending address), *sa* and *ea* are two special words of the saved file. If the file is not found (i.e., has not been saved), control returns to *altrtn*. The format of *array* is PSW1, PSW2, followed by registers R0 to R15.

4.8.2 SAVE: Storing a Load Module on the Disk

Model:	CALL SAVE (*file*,*array*,*sa*,*ea*)
Example:	CALL SAVE (FILE,USRVEC,SA,EA)

SAVE saves *sa* (starting address), *ea* (ending address) and *array*, an array consisting of PSW1, PSW2, and registers R0 to R15; followed by the core image which begins at *sa* and extends through *ea*. The information is placed on *file*.

APPENDIX A
GLOSSARY FOR DOS-32

command	what you type at the console; consists of a key word, a string of arguments, and a terminator.
file	a named collection of data; in DOS-32 there are many types of files, including data files, source-program files, object-program files, load-module files, and directories.
the system	DOS-32 itself; consists of many core-resident system routines, are called in FORTRAN CALL statements or as functions in FORTRAN expressions, according to specified calling sequences, which consist of a subroutine name followed by a parenthesized string of arguments, separated by commas.
system programs	the FORTRAN compiler, the MAC assembler, the loader, the editor, etc.; each system program is a single file.

user programs	programs written by the user; each user program is a single disk-resident file which is similar to the system programs but is loaded with a different command.
directory	a file which consists of a list of other files, their addresses, and other information about them; access to any file is through the directory in which it is listed.
MFD	master file directory in which are listed the MFD itself, each UFD, and various system files and system file directories.
UFD	directory in which all programs of a single user are listed; each user has his own UFD.
command directory	a special-purpose system file directory (named COMDIR) in which all system programs are listed.
FORTRAN object-time routines	subroutines which are required to perform certain standard FORTRAN services, such as I/O; calls to these subroutines are generated during compilation of a FORTRAN program; all FORTRAN object-time routines are stored in a library.

library	a single disk-resident file consisting of many routines, such as the FORTRAN object-time routines, math package, graphic routines, etc.; all libraries are listed in LIBDIR, a special-purpose system file directory.
file system	a collection of system routines and system tables which create and maintain files; all directories and files are part of the file system.
unit	a port to the I/O system; programs refer to units which have been associated with files, rather than to files themselves; a fixed number of these ports is available to each user, who may associate any file to which he has access with any available port.
device	physical storage medium and transporter of data.
new-line character	the ASCII character which indicates an end-of-line condition; corresponds to the linefeed key on a teletype and implies the dual linefeed/carriage-return function; unless preceded by the continuation character(;), also indicates the end of a command.

APPENDIX B
SUMMARY OF USER COMMANDS

Internal Commands

Model:	ATTACH *directory password* [*drive*]
Abbreviation:	A
Function:	Gives the user access to files in the specified *directory* on the specified *drive*
Model:	BINARY [(*directory password*)] *file*
Abbreviation:	B
Function:	Opens unit 3 to write an object file
Model:	CLOSE $\left\{\begin{array}{l}\textit{file}\\ \textit{unit} \ \ldots\\ \textit{ALL}\end{array}\right\}$
Abbreviation:	C
Function:	Closes all specified names and/or units
Model:	COMIPUT $\left\{\begin{array}{l}\textit{file}\\ \textit{TTY}\\ \textit{CONTINUE}\end{array}\right\}$
Abbreviation:	COMI
Function:	Allows the user to switch control from the console to a stored set of commands in a file
Model:	COMOUTPUT *file*
Abbreviation:	COMO
Function:	Allows the user to switch system responses to commands to some file

Model:	DELETE *file...*
Function:	Removes entries for the *files* from the current directory
Model:	INPUT [(*directory password*)] *file*
Abbreviation:	I
Function:	Connects *file* to unit 1
Model:	LISTFILE
Abbreviation:	LISTF
Function:	Types out the names of all files entered in the current directory
Model:	LISTING [(*directory password*)] *file*
Abbreviation:	L
Function:	Opens unit 2 to write a source listing
Model:	LOGIN *directory password*
Abbreviation:	LO
Function:	Attaches the user to the specified *directory* if the *password* matches
Model:	LOGOUT
Abbreviation:	LOGO
Function:	Closes any open files, units, and devices and detaches the user from the current directory
Model:	OPEN [(*directory password*)] *file unit key*
Abbreviation:	O
Function:	Opens the specified *unit* for a particular activity (identified by *key*) and associates with it a specified *file*.

Model:	PM
Abbreviation:	P
Function:	Types out the contents of the program status words and registers
Model:	RESTORE [(*directory password*)] *file*
Abbreviation:	REST
Function:	Reads the *file* into core storage
Model:	RESUME [(*directory password*)] *file* [psw_1]
Abbreviation:	R
Example:	RESUME (TSTONE) FORTEST 300
Function:	Reads *file* into core and initiates execution at the saved entry point or psw_1
Model:	SAVE *file starting-address ending-address* [*entry-point*]
Abbreviation:	SA
Function:	Creates a file called *file* from the contents of a defined portion of core
Model:	START [psw_1] [psw_2] [r_n]. . .
Abbreviation:	S
Function:	Initiates execution of the program in core
Model:	STARTUP $drive_1$ $drive_2$
Abbreviation:	STARTU
Function:	Establishes a sequence of logical *drives* in which the system will search in ATTACHing a directory.

External Commands

Model:	BEDIT
Function:	Invokes the binary edit program which allows inspection and manipulation of object-program files
Model:	BUGGY
Function:	Invokes the online debug package which allows users to examine the contents of any address, change those contents, perform address computation, and insert and remove break points
Model:	CARDFS
Function:	Creates a file from a card deck
Model:	CNAME [(*directory password*)] file_1 file_2
Function:	Changes the name of a file in a specified *directory*
Model:	CONCAT
Function:	Copies a set of named files into a single output file for use by the loader
Model:	COPYFS $\mathit{directory}_1$ [*drive*] $\mathit{directory}_2$ [NUM]
Function:	Copy one or more files from $\mathit{directory}_1$ on specified disk *drive* to $\mathit{directory}_2$ on drive 0. Adds sequence numbers to file if NUM is present.
Model:	CPRSAV
Function:	Compares two save files in dump format
Model:	DMPSAV
Function:	Lists a save file in dump format

Model:	EDIT [*file*]
Function:	Invokes the interactive text editor which allows users to create, edit, and store new files and to edit current files according to context
Model:	FTN [*option*]...
Function:	Invokes the FORTRAN compiler to compile a FORTRAN source program
Model:	LDR
Function:	Invokes the loader to load a file containing one or more object programs
Model:	LISTU [(*directory password*)] [*prefix*]
Function:	Creates two listing files in the specified *directory*
Model:	MAC [*option*]...
Function:	Invokes the MACRO assembler to assemble a MAC source program
Model:	MEDIA
Function:	Copies files of data from one storage medium to another
Model:	MOVEF *file* $directory_1$ $directory_2$
Function:	Inserts an entry for *file* in $directory_2$ and deletes its entry in $directory_1$

Model:	MOVEFS *directory*$_1$ *directory*$_2$
Function:	Moves several files from one ufd to another. See Section 3.6.4 for directions for use
Model:	PRINTFS
Function:	Lists source files in the line printer
Model:	SUPDATE
Function:	Invokes a noninteractive version of the text editor

APPENDIX C
COMMAND-NAME ABBREVIATIONS

DOS scans command-names as entered at the console and searches its internal list of commands for the first match. Each command can be abbreviated by truncating characters from the right, in such a way that the abbreviation is unique. Thus, BINARY can be abbreviated as B because no other command begins with the letter B. But when more than one command begins with the same letter, the abbreviations are less obvious. In these cases, the abbreviation must include enough characters to eliminate alternative interpretations of higher precedence, i.e., command-names which share the same letters and appear closer to the beginning of the command-name list (which is not arranged in alphabetical order).

Following is a list of DOS-32 user command-names and their minimum valid abbreviations.

Command	*Abbreviation*
ATTACH	A
ASSIGN	AS
BINARY	B
CLOSE	C
COMINPUT	COMI
COMOUTPUT	COMO
DELETE	DELETE
LISTING	L
LISTFILE	LISTF

Command	*Abbreviation*
LOGIN	LO
LOGOUT	LOGO
OPEN	O
PM	P
RESUME	R
RESTORE	REST
START	S
SAVE	SA
STARTUP	STARTU

APPENDIX D
PROGRAMMING HINTS AND WARNINGS

Miscellaneous Programming Information

1) The following codes are used by FORTRAN in the storage assignment table:

   # = TML$ - dimension, data

   % = Proc

   @ = TEM$ - other

   * = common

2) Check external list for names which shouldn't be there: misspellings or arrays not in dimension statement.

3) Check ALIST for names which shouldn't be there, especially @ variables: misspellings, failure to type, failure to initialize.

4) External names of the form X$nn (X = A,B,C,D; nn = 10,20,30,40; etc.) are type-conversion routines. Their appearance in the external list may indicate failure to type a variable or function properly.

5) Arguments to a subprogram are not affected by the IMPLICIT statement.

6) The shift operators (e.g., .LS.) have lower precedence than arithmetic operators, e.g., A + B .LS. 1 is interpreted as (A+B) .LS.1

7) The FORTRAN compiler causes all integers to be stored in what is called *integer position*. Integer position is one bit to the left of *normal position*. Hence the integer 3 would be stored as $00000006, while the integer -1 would be stored as $FFFFFFFE.

Efforts to frustrate FORTRAN in this regard <u>may</u> be sucessfully resisted by the compiler. In general, DOS has acquiesced in this matter. Therefore, in calling DOS routines,you should always expect to pass and receive any integers or single characters in integer position. Character strings with length greater than one are not put in integer position by either FORTRAN or DOS. Unfortunately, there is currently some confusion with regard to hexadecimal values, which are sometimes shifted to normal position by DOS before typing and sometimes not.

8) Note that certain function names will default to incorrect data types if you use them and fail to declare them in INTEGER or LOGICAL, while AND, OR, XOR, COMPL, and many others are INTEGER.

Temporary Discrepancies in the Manual and Known Bugs

1) Editor Problems

   a) If either a NEXT or a PRINT command has a negative argument which moves the pointer beyond the top of the file (TOP is typed out), the in-core buffer is lost. To continue you must QUIT and start over. Your file is not damaged.

2) FORTRAN Problems

   a) FTN occasionally stops in middle of a deck (DOS or OS-1). This is repeatable if tried immediately and usually goes away with time. It may be due to an uninitialized storage location in the dynamic area.

   b) Occasionally, a .NOT. in a logical expression is ignored. The problem goes away if the identical deck is recompiled.

The problem has been shown to depend upon the prior contents of the compiler work space.

c) An implied DO in a DATA statement may cause a storage variable to be improperly assigned to the TEM$ area with an offset appropriate for TM$.

d) The optimizer improperly assumes that arguments to a function cannot be changed by the function.

APPENDIX E
INITIALIZATION INSTRUCTIONS

The following instructions will enable you to start DOS-32, after which you will be able to use any of the commands presented in Chapter 3.

Readying the CPU: Various switches require special settings. The remainder should be placed in a neutral position (for those with three positions, the center position is neutral; otherwise, up is neutral).

ON-OFF key: turn to ON, then to UNLOCK.

O-F/ABS switch: set to ABS.

IOP SELECT switch: set IOP 1 down

DEVICE SELECT switch: set to 1011 (1 is down).

ACCESS/RUN/STEP switch: set to RUN

SENSE switches: all up.

Readying the disk: Mount the DOS-32 system pack on drive 0 (turn clockwise to mount, counter-clockwise to dismount). Depress the START/READY and PERMIT/PROTECT buttons and wait until START, READY and PERMIT are all lit.

Loading the system: Press SYSTEM, then press START. The following dialogue will result (user entries are in italics):

```
DOS-32 REV G
DATE=date
TIME=time
```

```
OK,LOGIN directory
OK,
   ...proceed with session...

OK, LOGOUT
OK
```

where *date* is an 8-character date, *time* is an 8-character time, and *directory* is the name of your UFD. If two drives are being used, the STARTUP 0 1 command should precede the LOGIN command.

APPENDIX F

DOS SYSTEM ROUTINES

The following system routines are available to the user. The user should be careful not to use the name of one of these routines for a user subroutine, or problems may occur.

LIST	T1IN
EXIT	T1OU
PRTERR	TNOU
GETERR	TNOUA
ERRRTN	TOHEX
ERRSET	
OPEN	
CLOSE	
BRWFLL	
REWIND	
DELETE	
ATTACH	
COMANL	
SAVE	
RESTOR	
RDASR	
WRASR	
CDRD	
PRINT	

APPENDIX G
OVERLAY STRUCTURES

A save file created by the loader (LDR) consists of a main program, all of the subroutines referenced by it, all of the defined common areas and local storage areas required, and copies of all library routines needed to resolve external references. The length of the save file is, therefore, the sum of the lengths of all of its component parts. When main storage is not at a premium, this is the most efficient form for program execution. However, if the size of a save file approaches the limits of the main storage available, the programmer should consider using an overlay structure.

The design of an overlay structure depends on the relationships among the subroutines and common areas within the program. Two subroutines that do not have to be in storage at the same time may overlay each other. Such subroutines are independent - that is, they do not reference each other either directly or indirectly. Subroutines are dependent if proper execution of one subroutine requires the presence of another (e.g., the routine which issued a call and to which control will be returned).

To design an overlay structure, the main program which receives control at the beginning of execution and all common areas which preserve data throughout execution are grouped together to form the root segment. The rest of the structure is developed by determining the dependencies among the remaining subroutines and common areas and then grouping them to form independent segments which can occupy overlapping areas of core at different stages during program execution. The procedure can best be illustrated by an example.

Assume that the purpose of the program is to perform a simulation. Program flow begins with the execution of subroutine INIT which reads parametric data describing the particular case to be simulated. The data are interpreted and converted to useful form and stored in two common areas, COMA and COMB. Then control is passed to subroutine SIM which conducts simulation beginning with an initial state described by parameters in COMB. As the simulation proceeds, raw event data are written to a disk file. Prior to termination, SIM stores summary data and status information in COMA. The data in COMB are no longer required. Control now passes to subroutine EDIT which examines the edit specifications placed in COMA by INIT. EDIT then scans the file of raw event data and produces an event summary before terminating.

INIT, SIM and EDIT are independent subroutines, but COMA and COMB must be shared. Assume further that subroutine RNDM is called by both INIT and SIM. An overlay structure which observes the relationships among all these modules can be represented as a tree as in Figure G.1.

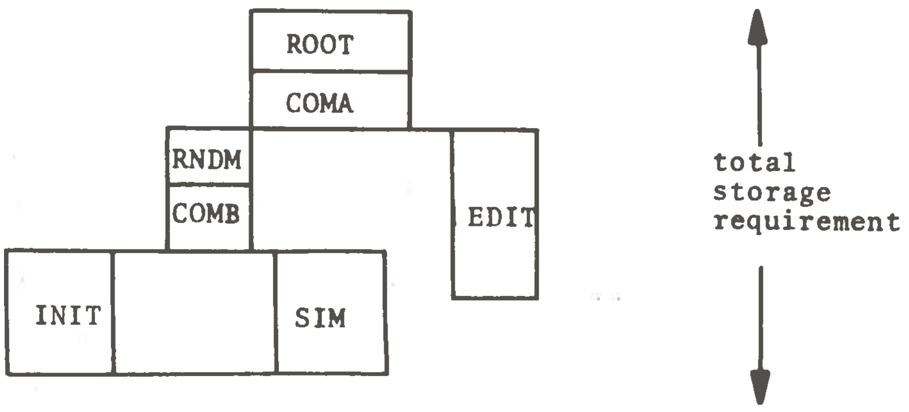

Figure G.1 Overlay Tree Structure

In the figure, the vertical dimensions of the rectangular areas are intended to represent, graphically, the storage requirements of the named modules. The total storage requirement is considerably less than would be required by a simple linear structure.

Using the facilities of the loader and the RESTOR subroutine, a group of save files can be created which will carry out the execution of the simulation. The following steps are required:

1) Compile a main program (ROOT) which includes a definition of COMA, and then a series of calls to pass control to INIT, SIM and EDIT as follows:

```
SUBROUTINE ROOT
COMMON/COMA/...
DIMENSION VEC (18)
CALL RESTOR ('INIT    ',VEC,0,0)
CALL INIT
CALL RESTOR ('SIM     ',VEC,0,0)
CALL SIM
CALL RESTOR ('EDIT    ',VEC,0,0
CALL EDIT
END
```

2) Include definitions of COMA in INIT, SIM and EDIT and of COMB in INIT and SIM.

3) After compiling all subroutines, invoke the loader as follows:

```
OK, LDR
 ≥  *,PRI,B:ROT,REF
 ≥  *,PRI,B:RNDM
 ≥  *,PRI,B:SIM,LIB,MAP
```

```
    DEF 00006400 COMA
    DEF 00006500 TML$
    DEF 00006600 ROOT
    DEF 00006800 TEM$
    DEF 00006900 TML$
    DEF 00006A00 RNDM
    DEF 00006B00 TEM$
    DEF 00006C00 COMB
    DEF 00006E00 TML$
    DEF 00007000 SIM
    DEF 00009000 TEM$
    DEF 00009 00 X
    DEF 00009300 Y
    DEF 00009500 Z
    REF 00000000 INIT 000066A0
    REF 00000000 EDIT 00006740
FUL=00009600
  ≥ SAVE,INIT,6900,9600,0200
  ≥ $6900,PRI,B:EDIT
    DEF 00006900 TML$
    DEF 00006B00 EDIT
    DEF 00008000 TEM$
    DEF 00008100 U
FUL=00008200
  ≥ SAVE,EDIT,6900,9500,0200
  ≥ SAVE.ROOT,6400,6900,6600
  ≥
  OK,
```

Four separate save files have now been created. When the module ROTT is executed, it will restore and call, in sequence, the modules INIT, SIM and EDIT.

The example presented above was selected because it is of sufficient complexity to reveal a number of complications. First, the longest branch of the tree was loaded first and the LIB option was deferred until the third load line. This was done to force the loading of library routine Y at a high address since it is called by both SIM and INIT.

Next, the SAVE command which creates the file SIM does not include COMB. At execution, INIT places data into COMB. If the SIM save file were to overlay COMB as well as INIT, the data would be lost. Note that although two REFs appear, neither is within the area SAVEd.

The next load line specifies an explicit load address for B:INIT, causing SIM to be overlaid. (SIM must be saved first.) Since the new FUL is below the Y entry point, the call to Y from INIT is correctly resolved.

The SAVE command to create the INIT module must include the RNDM subroutine since it will be overlaid by EDIT. It must also include the Y library module which is still intact. When SIM overlays INIT, RNDM will still be available. RNDM appears above INIT and SIM in the tree structure but functionally it is beneath them in the hierarchy of subroutine calls. This type of structure is frequently useful if two or more independent routines call a common subroutine.

Upon completion of SIM's execution, RNDM and COMB are no longer required. EDIT, therefore, may be loaded at a location immediately following the TEM$ area of ROOT. The loading of EDIT resolves the last REF - i.e., the call from ROOT to EDIT. Both EDIT and ROOT may now be saved.

The entry point addresses of the overlay segments should be specified as 0200 (the EXIT slot in the DOS transfer vector) since an overlay segment cannot be executed as a main program. If the ROOT routine contains calls to any of the library routines loaded following SIM, and if any such calls are executed prior to segment loading, then the entire area should be included in the SAVE, ROOT control line.

The loading of library routines may cause difficulty on occasion. Any subroutine which is overlaid during loading is no longer available for calls. Its entry point remains in the loader's symbol table and is used by the loader to resolve references. No warning is issued. Such problems can usually be resolved either by changing the order in which routines are loaded, or by using the force load line to add the entry name to the loader's symbol table at an earlier stage of the loading process.

APPENDIX H

FORTRAN ERROR MESSAGES

COMPILATION WARNING ERRORS

Code	Description
W00	Logical constant spelling or Block Name=Procedure Name
W01	Illegal mode mixing in expression
W02	Integer variable required
W03	Integer expression required
W04	DO spelling
W05	Undefined return point
W06	Duplicate statement number
W07	Illegal common name usage
W08	Illegal format array in I/O call
W09	Illegal use of statement number
W10	FORMAT statement number required
W11	Illegal DO termination
W12	Statement spelling error
W13	Illegal entry name usage
W14	Illegal entry parameter usage
W15	Function requires parameters
W16	Dummy appears in namelist
W17	Format statement requires statement number
W18	Format descriptor requires nonzero value
W19	Illegal format scale factor
W20	Format nest too deep
W21	Format descriptor requires value
W22	No path to this statement
W23	Subscript count in EQUIVALENCE incorrect
W24	Multiple equivalenced arrays
W25	Multiple common in an equivalence
W26	Common base lowered by an EQUIVALENCE statement
W27	Executable statements in BLOCK DATA subprogram
W28	Constant too large for field
W29	No RETURN in function or empty line with statement number
W30	No EOF after END or size statement not allowed
W31	Exponential mode error
W32	Relational IF in logical IF

COMPILATION TERMINATING ERRORS

Code	Description
T00	Illegal use of relational operator in expression
T01	Illegal procedure name
T02	Illegal operator for expression
T03	Improper constant
T04	Parenthesis count incorrect for parameter
T05	Illegal assignment form
T06	Illegal logical usage in expression
T07	Too many right parentheses
T08	Too many left parentheses
T09	Illegal parameter usage in expression
T10	Improper complex constant
T11	Unterminated relational
T12	Illegal unary operator usage
T13	Improper hexadecimal usage in Hollerith
T14	Illegal subscript syntax
T15	Must be integer variable or constant
T16	Backward DO reference
T17	Return in main program
T18	Nonstandard return in function
T19	Illegal assign syntax
T20	Illegal statement number
T21	Improper multi-character operator
T22	Improper character for syntax
T23	Program too large-exceeds 32,767 words
T24	Temporary storage region (TEM$) too large-exceeds 32,767 words
T25	COMMON block name is procedure name
T26	Name missing
T27	Illegal I/O END-ERR syntax
T28	Illegal I/O list name
T29	Improper I/O DO loop
T30	Illegal statement number syntax
T31	Illegal subscript syntax

Code	Description
T32	Must be integer constant
T33	Illegal array name usage
T34	Improper statement termination
T35	Illegal DO ordering
T36	Illegal EXPLICIT mode syntax
T37	Illegal IMPLICIT mode syntax
T38	Illegal statement syntax
T39	Illegal I/O unit number
T40	Illegal Hollerith constant
T41	Illegal CONNECT syntax
T42	Illegal namelist name
T43	Illegal statement function name
T44	Invalid Hollerith length in FORMAT statement
T45	Too many right parentheses in DATA statement
T46	Data cannot be in COMMON
T47	Illegal DO syntax in DATA statement
T48	Too many left parentheses in DATA statement
T49	Illegal data constant
T50	Insufficient data constants
T51	Too many data constants
T52	Data subscript variable not an implied DO variable
T53	Must be variable
T54	Illegal statement ordering
T55	Illegal in-line operator
T56	Illegal in-line register
T57	Illegal in-line value instruction
T58	Illegal in-line text
T59	Illegal in-line integer range
T60	Data pool overflow for expression
T61	Data pool overflow
T62	Invalid literal
T63	Common name usage
	Illegal subscript range
T64	DO in logical IF
T65	Logical IF in logical IF
T66	END in logical IF

RUN-TIME ERRORS

Code	Type of Error	Description
01	FORMAT	No leading '(' in FORMAT statement
02	FORMAT	Too many levels of parenthesis
03	FORMAT	Unrecognized separator in FORMAT statement
04	FORMAT	Missing subfield in FORMAT statement
05	FORMAT	Illegal parameter in T FORMAT descriptor
06	FORMAT	Improper '-' in FORMAT statement
07	FORMAT	Zero repeat count
08	FORMAT	Unrecognized descriptor in FORMAT statement
09	FORMAT	Field width missing
10	FORMAT	Zero field width
11	FORMAT	Missing '.' in D, E, F, G FORMAT fields
12	FORMAT	Missing field in FORMAT statement
13	FORMAT	I output of noninteger
14	FORMAT	L I/O of nonlogical variable
15	FORMAT	Unmatched ')'
16	FORMAT	FORMAT statement contains no argument fields to match variables in READ/WRITE statement
17	FORMAT	Exponent underflow on input
18	FORMAT	Exponent overflow on input
19	FORMAT	Exponent underflow on input (Same as 17)
20	FORMAT	Exponent overflow on input (Same as 18)
21	FORMAT	Illegal character within a field specified as numeric
22	FORMAT	Improper item size
23	FORMAT	I/O buffer exhausted
24	Not Used	
25	FORMAT	L input with item size not 1 word
26	FORMAT	A input with item size not 1 word
27	FORMAT	Illegal form for integer input
28	FORMAT	Integer overflow
29	FORMAT	Illegal hex digit in text output
30	NAMELIST	Illegal external name

Code	Type of Error	Description
31	NAMELIST	External name not in list
32	NAMELIST	Illegal terminator
33	NAMELIST	Illegal integer: subscript or repeat count
34	NAMELIST	Illegal dimension
35	NAMELIST	Illegal mode
36	NAMELIST	Illegal list
37		
38	Not Used	
39		
40	Status Check	File status
41	Status Check	End-of-File
42	Unformatted I/O	First record read not number 1
43	Unformatted I/O	Record smaller than argument list
44	BACKSPACE	Incorrect record control word
45	GO TO	GO TO index out of range
46	Device Check	Too many file names
47	Device Check	File type does not permit request (such as a READ on a line printer)

APPENDIX I
SUMMARY OF SYSTEM AND LIBRARY ROUTINES

The following nmenonics are used for variable names:

altrtn	alternate return address control transfers to in case of error; set by FORTRAN ASSIGN statement
array	normally an array of *words*
directory	eight letter (two word) directory name
file	eight letter (two word) filename
int	integer position variable
logical	variable of FORTRAN type LOGICAL
m	integer position counter
n	interger position counter
password	eight letter (two word) password name
pointer	variable contains address of argument
unit	integer position DOS unit (range 1-7)
word	normal position variable

System and library routines:

AND	(*worda, wordb*)
wordc =	AND (*worda,wordb*)
CALL	ATTACH (*directory,drive,password*,.TRUE.,*altrtn* gaining access to a directory
CALL	BRWFIL (*key,unit,array,n,rel,altrtn*) reading, writing, or manipulating a file

int	=	CHKSUM(*array,n*) computing checksum
CALL		CLOSE (*file-or-unit*) closing a file
rtnkey	=	COMANL (*key,array*) performing logical analysis of user commands
logical	=	COMEQV (*arraya,arrayb*) comparing command names
wordb	=	COMPL (*worda*) forming complement
CALL		DCNVRT (*int,array,n*) converting a number from decimal to ASCII
CALL		DELETE (*file,altrtn*) deleting a file
CALL		DOCARD (*array*) reading a card
CALL		DOPRIN (*array*) printing a line
CALL		EXIT returning to the system
CALL		GETERR (*array*) storing error information
word	=	GTHEX (*array,n,m*) search for hexadecimal number in character string and convert to hexadecimal
CALL		GTNAMS (*array,m,n,directory,password,file,flag, altret*) retrieving directory and file from a character string
pointer	=	GTREGS(0) retrieving the address of the register save area
CALL		HCNVRT (*word,array,n*) converting a word from hexadecimal to ASCII
int	=	LBIT (*array,n*) retrieving a bit
int	=	LCHAR (*array,n*) retrieving a character

int	=	LH (*word*) retrieving left-half of word
int	=	LHC (*pointer*) retrieving left-half of word, indirect
int	=	LHEX (*array,n*) retrieving a hexadecimal digit
int	=	LHWRD (*array,n*) retrieving a half-word
word	=	LIST (*pointer*) referencing an absolute core address
pointer	=	LOC (*word*) retrieving the address of a variable
word	=	MAKWRD (*inta,intb*) forming a word from two half-words
CALL		MT7IOC (*key,array,n,length,mode,tcu,mtt,status*) controlling magnetic tape
logical	=	NAMEQV (*arraya,arrayb*) comparing names
CALL		OPEN (*key,file,unit,altrtn,newfil,directory,password*) opening a file
wordc	=	OR (*worda,wordb*) forming logical sum
CALL		PNOU (*array,n*) punching a character string
CALL		PNOUB (*array,n*) punching a binary string
CALL		PRTERR typing an error message
CALL		PUTC (*int,array*) inserting a character in an array
CALL		P1IN (*int*) reading a character from paper tape
CALL		P1INB (*int*) reading a binary character
CALL		P1OU (*int*) punching a character
CALL		P1OUB (*int*) punching a binary character

CALL	RDASR (*array,n*) reading a line from the teletype
CALL	RESTOR (*file,array,0,altrtn*) bringing a load module into core
CALL	REWIND (*unit,altrtn*) repositioning a unit to the beginning of a file
int =	RH (*word*) retrieving right-half of word
int =	RHC (*pointer*) retrieving right half of word, indirect
int =	RT (*n, word*) retrieving an integer
CALL	SAVE (*file,array,sa,ea*) storing a load module on the disk
CALL	SBIT (*array,n,int*) storing a bit
CALL	SCHAR (*array,n,int*) storing a character
CALL	SHWRD (*array,n,int*) storing a half-word
CALL	SLH (*word,int*) storing number in left-half of word
CALL	SLHC (*pointer, int*) storing number in left-half of word, indirect
CALL	SRH (*word,int*) storing number in right-half of word
CALL	SRHC (*pointer,int*) storing number in right-half of word indirect
logical =	SSWS (*n*) testing a sense switch
CALL	TNOU (*array,n*) typing a character string followed by newline on the teletype

CALL	TNOUA (*array,n*) typing a character string on the teletype
CALL	TOHEX (*word*) converting a word to hexadecimal and typing it
CALL	T1IN (*int*) reading a character from the teletype
CALL	T1OU (*int*) typing a character on the teletype
CALL	WRASR (*array,n*) typing a line on the teletype
wordc	= XOR (*worda,wordb*) forming exclusive OR

APPENDIX J
FORTRAN LIBRARY

The FORTRAN Library (LIBRARY in directory LIBRIR) has been modified from the OS-1 supplied library. Subroutines modified are BAC, DCK, END, PAU, STO, REF, WRF, REU, W$RU, and R$EW. A new routine SETST was written to dummy a status word. Furthermore, all the routines listed in section 4.5 Utility Functions, and section 4.6 Peripheral Device Routine were added to the library.