PRIME

THE BASIC/VM PROGRAMMER'S GUIDE PDR3058

P/N MAN3251-001

PRIME SOFTWARE DOCUMENTATION SUMMARY

Description	Software Rev. #	Document Number	Price
FORTRAN			
• The FORTRAN Programmer's Guide			
Bound edition	16	FDR3057-101A†	$15.00
Loose-leaf edition	16	FDR3057-101B†	$15.00
• The FORTRAN Programmer's Companion	16	FDR3338†	$ 2.00
COBOL			
• The COBOL Programmer's Guide	16	PDR3056†	$15.00
RPGII			
• The RPGII Programmer's Guide	16	PDR3031†	$15.00
• The RPGII Debugging Template	14-16	FDR3275	$ 2.00
BASIC/VM (COMPILED)			
• The BASIC/VM Programmer's Guide	16	PDR3058†	$15.00
• The BASIC/VM Programmer's Companion	16	FDR3341†	$ 2.00
BASIC (INTERPRETIVE)			
• The Interpretive BASIC Programmer's Guide	14,15	IDR1813	$15.00
Technical update	16	PTU59†	$ 2.00
ASSEMBLY LANGUAGE			
• The Assembly Language Programmer's Guide			
Bound edition	16	FDR3059-101A†	$15.00
Loose-leaf edition	16	FDR3059-101B†	$15.00
• The Assembly Language Programmer's Companion	15,16	FDR3340	$ 2.00
• The System Architecture Reference Guide	16	PDR3060†	$15.00
PRIMOS OPERATING SYSTEM/UTILITIES			
• The PRIMOS Commands Reference Guide			
Bound edition	16	FDR3108-101A†	$15.00
Loose-leaf edition	16	FDR3108-101B†	$15.00
• The PRIMOS Commands Programmer's Companion	16	FDR3250†	$ 2.00
• The System Administrator's Guide	16	PDR3109†	$15.00
• The System Administrator's Programmer's Companion	16	FDR3622†	$ 2.00
• The New User's Guide to EDITOR and RUNOFF			
Bound edition	15	FDR3104-101A	$15.00
Loose-leaf edition	15	FDR3104-101B	$15.00
Change sheet update	16	COR3104-001†	$ 3.00
• PRIMOS Subroutines Reference Guide	16	PDR3621†	$15.00
• LOAD and SEG Reference Guide	16	IDR3524†	$15.00
DATA MANAGEMENT			
• DBMS Administrator's Guide	16	PDR3276†	$15.00
• DBMS Schema Reference Guide	16	PDR3044†	$15.00
• DBMS FORTRAN Reference Guide	16	PDR3045†	$15.00
• DBMS COBOL Reference Guide	16	PDR3046†	$15.00
• The PRIME/POWER Guide	16	IDR3709	$15.00
• The MIDAS Reference Guide	14	IDR3061	$15.00
Technical update	16	PTU60†	$ 2.00
• The FORMS Programmer's Guide	16	PDR3040†	$15.00
STATISTICS			
• The SPSS Programmer's Guide	16	PDR3173†	$15.00
COMMUNICATIONS			
• The PRIMENET Guide	16	IDR3710†	$15.00
• The RJE/2780 Guide	16	PDR3067†	$15.00
• The HASP Guide	16	PDR3107†	$15.00
• The [illegible]200 Guide	16	IDR3431†	$15.00
SYS[illegible] INSTALLATION			
• The System Installer's Guide	15	PDR3105†	$15.00

†-Denotes new or revised title

PRIME'S BASIC/VM PROGRAMMER'S GUIDE (REV. Ø, April 1979)

This guide documents Prime BASIC/VM and the relevant PRIMOS operating system features as implemented at Master Disk Revision Level 16. Information appearing in the Rev. 14 IDR and PTU57 (Rev. 15), has been corrected as necessary and incorporated into the text of this guide.

This guide is organized to make life easier for you, the BASIC/VM application programmer or casual user.

We assume you are familiar with BASIC in some form, and will easily adapt to Prime's implementation and extensions, which are fully defined in this guide.

Most PRIME system users will want to know something about our operating system, PRIMOS. Deciding which of the many PRIMOS concepts are important in BASIC/VM programming is not an easy task. Therefore, this guide contains a summary of the most useful PRIMOS concepts, terms and commands. Should additional details be required to complete special programming tasks, we refer you to the appropriate PRIME documents in which supplemental information can be found.

The result is a single document containing everything you need to know to write, modify, compile, execute, and debug most BASIC/VM application programs.

We hope you will find this a helpful guide to the particulars of BASIC/VM programming within the PRIMOS operating system. We invite comments on the organization and philosophy of this guide, as well as its contents, accuracy, and clarity.

All correspondence on suggested changes to this document should be directed to:

Laura Douros, Technical Writer
Technical Publications Department
Prime Computer, Inc.
145 Pennsylvania Avenue
Framingham, Massachusetts 01701

Acknowledgements:

We wish to thank the members of the BASIC/VM PROGRAMMER'S GUIDE team and also the non-team members, both customer and Prime, who contributed to and reviewed this PDR.

First Printing April 1979

TABLE OF CONTENTS

Section	Title	Page
SECTION 1	INTRODUCTION TO BASIC/VM	1-1
	AUDIENCE	1-1
	DESCRIPTION OF BASIC/VM	1-1
	BASIC/VM FEATURES	1-2
	HOW TO USE THIS MANUAL	1-2
	COMPATIBILITY WITH OTHER FORMS OF BASIC	1-3
SECTION 2	OVERVIEW OF PRIMOS	2-1
	INTRODUCTION TO PRIMOS	2-1
	CONVENTIONS	2-2
	USING THE FILE SYSTEM	2-5
	ACCESSING THE SYSTEM (LOGIN)	2-11
	DIRECTORY OPERATIONS	2-12
	SYSTEM INFORMATION	2-15
	FILE OPERATIONS	2-18
	LOGOUT	2-22
SECTION 3	USING BASIC/VM	3-1
	ENTERING BASICV SUBSYSTEM	3-1
	USING BASIC/VM COMMANDS	3-2
	EXITING BASICV SUBSYSTEM	3-13
	ACCESSING FILES IN REMOTE DIRECTORIES	3-14
	RUNNING PROGRAMS FROM PRIMOS	3-16
	MODES OF OPERATION IN BASICV	3-17
SECTION 4	ELEMENTS OF BASIC/VM	4-1
	LANGUAGE ELEMENTS	4-1
	OPERANDS	4-1
	CONSTANTS	4-2
	VARIABLES	4-3
	FUNCTIONS	4-4
	OPERATORS	4-5
	EXPRESSIONS	4-7
	COMMANDS AND STATEMENTS	4-7
	STATEMENT SYNTAX	4-8
	COMMENTS	4-9
	LIST OF COMMANDS	4-10
	LIST OF STATEMENTS	4-11

SECTION 5 DATA INPUT/OUTPUT 5-1

DATA INPUT STATEMENTS 5-1
DATA INPUT FROM THE TERMINAL 5-3
DATA OUTPUT STATEMENTS 5-6
DEFAULT PRINTING 5-6
PRINTING WITH MODIFIERS 5-7
FORMATTING WITH PRINT USING 5-10

SECTION 6 PROGRAM CONTROL STATEMENTS 6-1

INTRODUCTION 6-1
STATEMENT MODIFIERS 6-2
BRANCHING WITHIN A PROGRAM 6-2
BRANCHING TO EXTERNAL PROGRAMS 6-4
CONDITIONAL PROGRAM BRANCHING 6-8
LOOP STATEMENTS 6-11

SECTION 7 EDITING AND DEBUGGING 7-1

INTRODUCTION 7-1
EDITING BASIC/VM PROGRAM 7-1
DEBUGGING A PROGRAM 7-6
TRAPPING EXECUTION ERRORS 7-10

SECTION 8 FILE HANDLING 8-1

INTRODUCTION 8-1
OPENING A DATA FILE 8-2
ACCESS METHODS 8-6
SAM FILE HANDLING 8-8
ADDING DATA TO SAM FILES 8-8
READING SAM FILES 8-10
TRAPPING ERRORS 3-16
CLOSING DATA FILES 8-17
DAM FILE HANDLING 8-18
WRITING DATA TO DAM FILES 8-19
READING DAM FILES 8-21
SEGMENT DIRECTORIES 8-22
MIDAS FILE ACCESS 8-24

SECTION 9 ARRAY AND MATRIX MANIPULATIONS 9-1

ARRAYS, MATRIX INTRODUCTION 9-1
DEFINING ARRAYS 9-2
MATRIX OPERATIONS 9-5
MATRIX I/O 9-15
MATRIX - DATA FILE I/O 9-17

SECTION 10 NUMERIC AND STRING FUNCTIONS 10-1

NUMERIC SYSTEM FUNCTIONS 10-1
STRING SYSTEM FUNCTIONS 10-8
USER-DEFINED FUNCTIONS 10-2

SECTION 11 NUMERIC DATA 11-1

NUMERIC CONSTANTS 11-1
NUMERIC VARIABLES 11-2
NUMERIC ARRAYS, MATRICES 11-3
NUMERIC EXPRESSIONS 11-3
EVALUATING EXPRESSIONS 11-5
NUMERIC OPERATORS 11-6

SECTION 12 STRING DATA 12-1

STRING CONSTANTS 12-1
STRING VARIABLES 12-1
STRING ARRAYS, MATRICES 12-2
STRING EXPRESSIONS 12-3
STRING OPERATORS 12-4
EVALUATING STRING EXPRESSIONS 12-5

SECTION 13 PRIMOS SYSTEM COMMANDS 13-1

ALPHABETICAL LIST OF PRIMOS COMMANDS 13-1

SECTION 14 BASIC/VM SYSTEM COMMANDS 14-1

ALPHABETICAL LIST OF BASIC/VM SYSTEM COMMANDS 14-1

SECTION 15 BASIC/VM STATEMENTS 15-1

ALPHABETICAL LIST OF BASIC/VM STATEMENTS 15-1

APPENDIX A SAMPLE PROGRAMS A-1

THREE SAMPLE BASIC/VM PROGRAMS

APPENDIX B ASCII CHARACTER SET B-1

APPENDIX C RUN-TIME ERROR CODES C-1

LIST OF BASIC/VM RUN-TIME ERROR MESSAGES C-1

APPENDIX D ADDITIONAL PRIMOS FEATURES D-1

GLOSSARY OF PRIME CONCEPTS AND TERMS D-1
SETTING TERMINAL CHARACTERISTICS D-7
SUMMARY OF EDITOR COMMANDS D-8
SUMMARY OF COMMAND FILE FEATURES D-11

APPENDIX E ADVANCED FILE HANDLING E-1
ACCESS METHODS E-1
DATA STORAGE PATTERNS E-2
ACCOMODATING LARGE DATA ITEMS E-7
READING ASCII FILES E-11

ILLUSTRATIONS AND TABLES

ILLUSTRATIONS

2-1	Examples of Files and Directories in PRIMOS Tree-Structured File System	2-8
3-1	Process of a BASIC/VM Program	3-9
8-1	File Structures	8-6
8-2	Deleting a SEGDIR Data File	8-40
8-3	Configuration of MIDAS File	8-42
9-1	Matrix Addition	9-9
9-2	Matrix Inverse	9-14
11-1	Logical Expressions	11-9

TABLES

2-1	Types of Files in PRIMOS	2-7
2-2	Useful System Information	2-16
4-1	Legal and Illegal Variables	4-4
4-2	List of Commands and Statements	4-10
8-1	File Type-Codes	8-4
9-1	Matrix Operations	9-6
10-1	Numeric System Functions	10-2
10-2	String System Functions	10-9
10-3	Masks for CVT$$	10-10
11-1	Numeric Operators	11-5
15-1	File Type-Codes	15-5
15-2	Numeric Format Field Characters	15-21
15-3	String Format Field Characters	15-23

SECTION 1

INTRODUCTION TO BASIC/VM

AUDIENCE

This document has been prepared for the BASIC user or programmer who is not acquainted with Prime's BASIC/VM. It is recommended that those unfamiliar with any form of the BASIC language refer to a commercially available text. For example:

Marateck, Samuel, BASIC; Academic Press, Inc.

Waite and Mather, Editors, BASIC, Sixth Edition; University Press of New England.

This document defines the Prime BASIC/VM language and illustrates its major uses with many examples. In addition, it introduces Prime's operating system, PRIMOS, and enables new users to access and use the system.

DESCRIPTION OF BASIC/VM

Prime's BASIC/VM, or virtual-memory BASIC, is a high level problem-solving language useful in research, business, and educational facilities. Its simple and easily understood language structure makes it suitable for writing programs to handle both simple and complicated mathematical problems. The language consists of system commands which are directives to the BASIC/VM subsystem, and statements, which are the fundamental components of programs. BASIC/VM programs are composed of numbered statements and optional comments, which are notations to the user.

BASIC/VM is a compiler for an extended form of the standard BASIC language. It is an upward compatible extension of Prime's BASIC Interpreter, employing the fast program execution and virtual memory capabilities of the Prime 35Ø and higher central processors. Programs previously written in interpretive BASIC will run under BASIC/VM without modification.

The BASIC/VM Language Processor

The components of the BASIC processor are:

- BASIC Language Compiler
- Command Processor
- Statement Editor

The command processor interprets and executes all system level directives. The language compiler translates program source code into

executable machine language. The statement editor enables complete modification of BASIC/VM programs on a line-by-line basis entirely within the BASICV system.

Features

There are a number of features which distinguish BASIC/VM from other forms of the BASIC language. These include:

1. The ability to support multiple users without significant performance loss.

2. It is compiled, rather than interpreted, for rapid program execution.

3. The ability to run very large programs without compromising small program efficiency.

4. The ability to access MIDAS keyed-index files.

5. Special methods of formatting output.

6. Extensive alphanumeric string support.

7. Matrix manipulations.

8. Multiple data segments and 128KB (64K word) procedure space.

9. Immediate mode for instant calculations.

1Ø. An editor for complete program modification.

11. A library of mathematical functions to aid in calculations.

12. Recursive function capability.

13. Extended control functions (DO, DOEND, etc.) and statement modifiers (WHILE, UNTIL, UNLESS, FOR, IF).

14. Double-precision floating-point numeric data.

HOW TO USE THIS MANUAL

This manual has been organized to accommodate several levels of user experience. If you know BASIC in some form and have used it on other systems but not Prime's, read Section 2 to familiarize yourself with Prime's operating system. A discussion of the terms and conventions appearing in this manual is located in the first part of Section 2. More information on Prime's embedded file management system (FMS), useful PRIMOS features (e.g., EDITOR), as well as PRIMOS-related terminology, is found in Appendix D.

If you have previously used a Prime computer and do not need a review of system access and file manipulation, proceed to Section 3, which describes the important commands and concepts you need to work with BASIC/VM. It might be helpful to familiarize yourself with the conventions and terms in Section 2, however.

A capsule summary of the BASIC/VM language elements, including a complete list of commands and statements, is found in Section 4. Programmers who have previously used a Prime system and/or Prime's Interpretive BASIC may need only Section 4 to get started.

Basic programming information, including details on program control structure, data transfer, file handling, editing and debugging a program, is discussed in Sections 5 through 8. Section 9 deals with matrix and array manipulations. Section 1Ø contains a library of all numeric and string system functions and describes how to define and implement user-defined functions. Sections 11 through 15 comprise a complete reference to the BASIC/VM language, including numeric and string data, commands and statements.

The Appendices contain run-time error messages, ASCII character set list, glossary of PRIMOS concepts and terms, an overview of useful PRIMOS features, (e.g. EDITOR, Command Files), and sample programs.

Compatibility with Other Forms of BASIC

While Prime BASIC/VM is generally compatible with other versions of BASIC, users should be aware of the following restrictions and alternative implementation features:

- The ability to enter several statements on one line is not supported.
- There is no 'BYE' command. 'QUIT' is its equivalent in BASIC/VM.
- No lowercase input is accepted on command lines.
- The double-quote character is not accepted by BASIC/VM as a delimiter as it is the default PRIMOS erase character.(See Section 2, Conventions.) Single quotes are used as delimiters, e.g., 'This is OK'.
- Prime's BASIC supports the use of statement modifiers like WHILE, UNTIL, UNLESS with statements like IF, PRINT and FOR-NEXT loops.
- Prime's BASIC/VM supports double-precision floating-point numeric data.
- The assignment statement LET is optional in BASIC/VM.

- BASIC/VM has control features like two-branch deciders, e.g., IF...THEN...ELSE, and logical loop control via the modifiers WHILE, UNTIL, UNLESS, which allow the writing of structured language.

SECTION 2

OVERVIEW OF PRIMOS

INTRODUCTION TO PRIMOS

This section introduces Prime's operating system, PRIMOS, and its most commonly used commands. All users of BASIC/VM will need some of the information here to access the system and work with files. The remainder of the information is necessary to accomplish more advanced programming tasks. Users new to Prime's system are referred to the glossary of terms in Appendix D.

Additional information on all the subjects discussed in this section may be found in the following Prime documents:

New User's Guide To Editor And Runoff

PRIMOS Programmer's Companion

Reference Guide, PRIMOS Commands

Subroutine Reference Guide

CONVENTIONS

The conventions for PRIMOS commands are:

- WORDS-IN-UPPER-CASE

Capital letters identify command words or keywords. They are to be entered literally. If a portion of an upper-case word is underlined, the underlined letters indicate the minimum legal abbreviation.

- Words-in-lower-case

Lower case letters identify parameters. The user substitutes an appropriate numerical or text value.

- Braces { }

Braces indicate a choice of parameters and/or keywords. Unless the braces are enclosed by brackets, at least one choice must be selected.

- Brackets [ ]

Brackets indicate that the word or parameter enclosed is optional.

- Hyphen -

A hyphen identifies a command line option, as in: SPOOL -LIST

- Parentheses ( )

When parentheses appear in a command format, they must be included literally.

- Ellipsis ...

The preceding parameter may be repeated.

- Angle brackets < >

Used literally to separate the elements of a pathname. For example: <FOREST>BEECH>BRANCH537>TWIG43>LEAF4.

- option

The word option indicates one or more keywords or parameters can be given, and that a list of options for the particular command follows.

- Spaces

Command words, arguments and parameters are separated in command lines by one or more spaces. In order to contain a literal space, a

parameter must be enclosed in single quotes. For example, a pathname may contain a directory having a password:

'<FOREST>BEECH SECRET>BRANCH6'.

The quotes ensure that the pathname is not interpreted as two items separated by a space.

SPECIAL TERMINAL KEYS

- CONTROL

The key labeled CONTROL (or CTRL) changes the meaning of alphabetic keys. Holding down CONTROL while pressing an alphabetic key generates a Control Character. Control characters do not print. Some of them have special meanings to the computer. Others are ignored.

- RETURN

The RETURN key ends a line. PRIMOS edits the line according to any Erase (") or Kill (?) characters, and either processes the line as a PRIMOS command, or passes it to a utility such as the editor. RETURN is also called CR or CARRIAGE-RETURN.

- BREAK

See CONTROL-P.

Special Characters

- Caret (^)

Used in EDITOR to enter octal numbers and for literal insertion of Erase and Kill characters. On some terminals and printers, prints as up-arrow (↑).

- Backslash (\)

Default EDITOR tab character.

- Double-quote (")

Default erase character for PRIMOS, EDITOR, and RUNOFF Command Mode. Each double-quote erases a character from the current line. Erasure is from right (the most recent character) to left. Two double quotes erase two characters, three erase three, and so forth. You cannot erase beyond the beginning of a line. The PRIMOS command TERM (See Appendix D of this guide) allows the user to choose a different erase character.

- Question mark (?)

Default kill character for PRIMOS, EDITOR, and RUNOFF Command Mode. Each question mark deletes all previous characters on the line. The PRIMOS command TERM (Appendix D of this guide) allows the user to choose a different kill character.

- CONTROL-P

QUIT immediately (interrupt/terminate) from execution of current command and return to PRIMOS level. Echoes as QUIT. Used to escape from undesired processes. Will leave used files open in certain circumstances. Equivalent to hitting BREAK key.

- UNDERSCORE (_)

On some devices, prints as a backarrow (<-).

SYSTEM PROMPTS

The OK Prompt

The OK prompt indicates that the most recent command to PRIMOS has been successfully executed, and that PRIMOS is ready to accept another command from the user.

PRIMOS supports type-ahead. The user need not wait for the "OK," after one command before beginning to type the next command. However, since each character echoes as the user types it, output from the previous command may appear on the terminal to be jumbled with the command being typed ahead. Type ahead is limited to 192 characters.

The ER! Prompt

The ER! prompt indicates that PRIMOS was unable to execute the most recent command, for one reason or another, and that PRIMOS is ready to accept another command from the user. The ER! prompt usually is preceded by one or more error messages indicating what PRIMOS thought the trouble was.

Common errors include:

- Typographical errors
- Omitting a password
- Being in the wrong directory
- Forgetting a parameter or argument

USING THE FILE SYSTEM

File and Directory Structures

A PRIMOS file is an organized collection of information identified by a filename. The file contents may represent a source program, an object program, a run-time memory image, a set of data, a program listing, text of an on-line document, or anything the user can define and express in the available symbols.

Files are normally stored on the disks attached to the computer system. No detailed knowledge of the physical location of a file is required because the user, through PRIMOS commands, refers to files by name. On some systems, files may also be stored on magnetic tape for backup or for archiving.

PRIMOS maintains a separate user file directory (UFD) for each user to avoid conflicts that might arise in assignment of filenames. A master file directory (MFD) is maintained by PRIMOS for each logical disk connected to the system. The MFD contains information about the

location of each User File Directory (UFD) on the disk. In turn, each UFD contains information about the location and content of each file or sub-UFD in that directory.

The types of files most often encountered are shown in Table 2-1. For a primer on the PRIMOS file system and a description of the ordering of information within files, refer to the Subroutine Reference Guide.

Directory Structure

The PRIMOS file directory system is arranged as a tree. At the root are the disk volumes (also called partitions, or logical disks). Each disk volume has a Master File Directory (MFD) containing the names of User File Directories (UFDs). Each UFD may contain not only files, but subdirectories (sub-UFDs), and they may contain subdirectories as well. Directories may have subdirectories to any reasonable level.

Pathnames

A pathname (also called a treename) is a name used to specify uniquely any particular file or directory within PRIMOS. It consists of the names of the disk volume, the UFD, a chain of subdirectories, and the target file or directory. For example,

```
<FOREST>BEECH>BRANCH5>SQUIRREL
```

specifies a file on the disk volume FOREST, under the UFD BEECH and the sub-UFD BRANCH5. The file's name is SQUIRREL. Figure 2-1 illustrates how pathnames show paths through a tree of directories and files.

Disk volume names, and the associated logical disk numbers, may be found with the STATUS DISKS command, described later. A pathname can be made with the logical disk number, instead of the disk volume name. For example, if FOREST is mounted as logical disk 3,

```
<3>BEECH>BRANCH5>SQUIRREL
```

specifies the same file as the previous example.

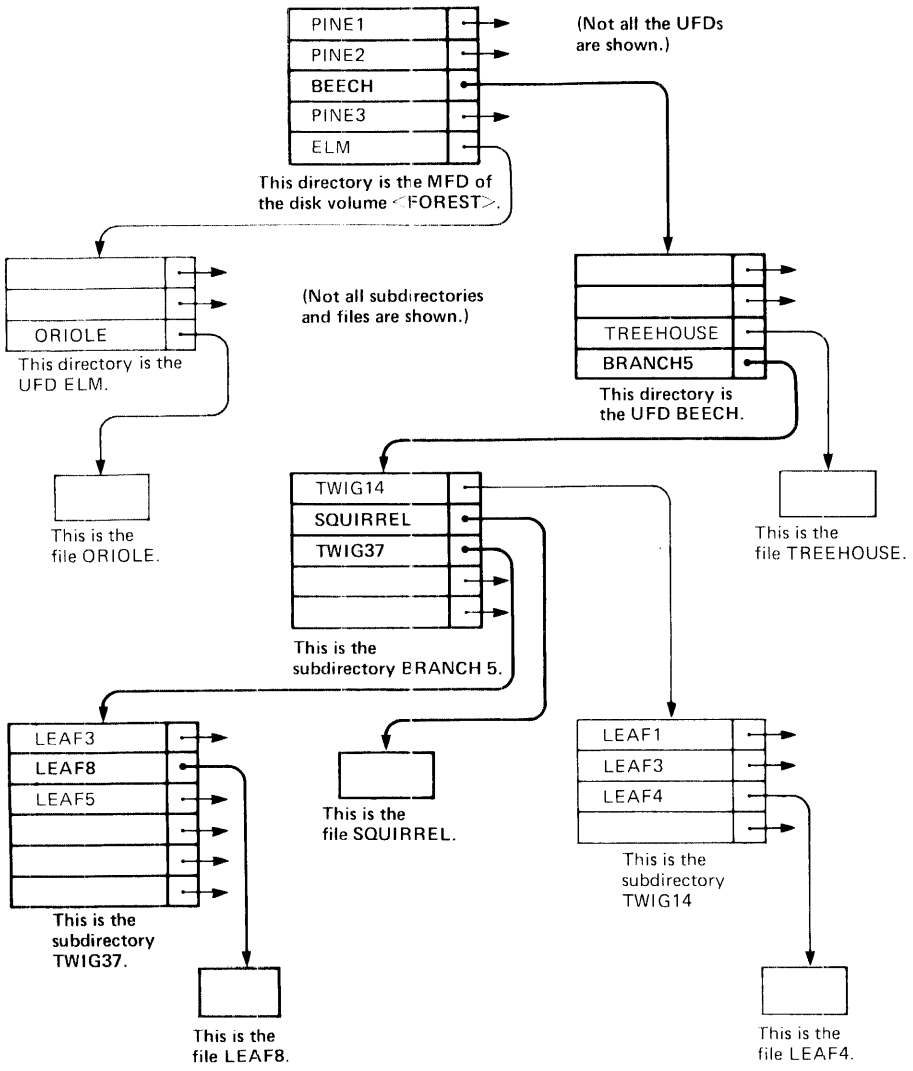
Usually each UFD name is unique throughout all the logical disks. In our example that would mean that there would be only one UFD named BEECH in all the logical disks, Ø through 17. When that is the case, the volume or logical disk name may be omitted, and PRIMOS will search all the logical disks, starting from Ø, until the UFD is found. For example, if there is no UFD named BEECH on disks Ø, 1, or 2, then

```
BEECH>BRANCH5>SQUIRREL
```

will specify the same file as the previous two examples. This last form of pathname, in which the disk specifier is omitted, is called an ordinary pathname because it is very frequently used.

Table 2-1.
Types of Files in PRIMOS

File Type	How Created	How Accessed	How Deleted	Use
ASCII, uncompressed	Programs SORT COMOUTPUT	Programs ED (examine only) SLIST SPOOL FTN READ/WRITE	DELETE FUTIL DELETE	Source files, text, data records for sequential access
ASCII, Compressed	CMPRES Some COBOL programs, ED	EXPAND to ASCII SPOOLer with EXPAND option ED	DELETE FUTIL DELETE	Same as uncompressed ASCII
Object (Binary)	Translators:RPG, FTN, PMA, COBOL, Binary Editor	LOAD or SEG Binary Editor (EDB)	DELETE FUTIL DELETE Binary Editor	Input to SEG or LOAD, Libraries
Saved Memory Image	LOAD Applications programs	TAP, PSD Control panel	DELETE FUTIL DELETE	Runfiles
Segmented runfile	SEG	SEG, VPSD Control panel	SEG DELETE FUTIL TREDEL	Runfiles
Segmented data file	SGDR$$ subroutine MIDAS, DBMS	SGDR$$ subroutine MIDAS DBMS	FUTIL TREDEL MIDAS KIDDEL	Data records for direct access
UFD Sub-UFD	CREATE	Contents: LISTF	DELETE FUTIL TREDEL	Used by PRIMOS
MFD	MAKE	Contents: LISTF	NO	Used by PRIMOS
Disk record availability table DSKRAT file	MAKE	NO	NO	Used by PRIMOS
BOOT	MAKE	NO	NO	Used by PRIMOS
CMDNCØ	MAKE	Contents: LISTF	NO	Used by PRIMOS



Pathnames identify files in a tree-structured file system. The pathnames:
<FOREST>BEECH>BRANCH5>SQUIRREL
<FOREST>BEECH>BRANCH5>TWIG37>LEAF8
are illustrated.

Figure 2-1. Examples of Files and Directories in PRIMOS Tree-structured File System.

Pathnames vs. Filenames

Most commands accept a pathname to specify a file or a directory; the terms "filename" and "pathname" may be used almost interchangeably. A few commands, however, require a filename, not a pathname. It's easy to tell a filename from a pathname. A pathname always contains a ">", while a filename or directory name never does.

Home vs. Current Directories

PRIMOS has the ability to remember two working directories for each user: the "home" directory, and the "current" directory. With few exceptions, the home and current directories are the same. All work can be accomplished while treating them both under the single concept of "working directory".

When the user logs in to a UFD, that UFD becomes the working directory. The ATTACH command changes the working directory to any other directory to which the user has access rights. A working directory may be an MFD, UFD, or sub-UFD.

The ATTACH command has a home-key option which allows the current directory to change while the home directory remains the same. See Reference Guide, PRIMOS Commands for details of this operation.

Relative pathnames

It is often more convenient to specify a file or directory pathname relative to the home directory, rather than via a UFD. For example, when the home directory is

```
BEECH>BRANCH5
```

the commands

```
OK, SLIST BEECH>BRANCH5>TWIG9>LEAF3
```

and

```
OK, SLIST *>TWIG9>LEAF3
```

have the same meaning. The symbol "*" as the first directory in a pathname means "home directory".

Current disk

Occasionally it will be necessary to specify a UFD on the disk volume you are currently using, that is, where your home directory is. For example, when developing a new disk volume with UFD names identical to

those on another disk, it is necessary to carefully specify which disk is to be used each time a pathname is given. The current disk is specified by

```
<*>BEECH>BRANCH5
```

for example. Do not confuse "<*>", meaning current disk, with "*>", which means home directory.

SYSTEM ACCESS

Introduction

The remainder of this section is a brief overview of some of the fundamental features of the PRIMOS operating system. It assumes that you are a BASIC/VM programmer with previous experience on an interactive computer system, although possibly not on a Prime computer. It also assumes that you have read the concepts and definitions in Appendix D, or that you are already familiar with PRIMOS terms. The commands introduced here allow you to:

- Gain admittance to the computer system (LOGIN).
- Change the working directory (ATTACH).
- Create new directories for work organization (CREATE).
- Secure directories against intrusion (PASSWD).
- Remove directories which are no longer needed (DELETE).
- Examine the location of the working directory and its contents (LISTF).
- Look at the availability and current usage of system resources - space, users, etc. (AVAIL, STATUS, USERS).
- Create files at the terminal (also see Appendix D, EDITOR).
- Rename files (CNAME).
- Determine file size (SIZE).
- Examine files (SLIST).
- Print files (SPOOL).
- Remove unneeded files (DELETE).
- Allow controlled access to files (PROTEC).
- Complete a work session (LOGOUT).

ACCESSING THE SYSTEM

In order to access or work in the system, the user must first follow a procedure known as 'login'. 'Logging in' identifies the user to the system and establishes the initial contact between system and user (via a terminal). Once logged in, the user has access to a working directory (work area), to files and to other system resources. The

format of the LOGIN command is:

LOGIN ufd-name [password] [-ON nodename]

ufd-name	The name of your login directory.
password	Must be included if the directory has a password.
-ON nodename	Used for remote login across PRIMENET network.

Example:

```
LOGIN DOUROS NIX
DOUROS (21) LOGGED IN AT 1Ø'33 112878
```

The number in parentheses is the PRIMOS-assigned user number (also called 'job' number). The time is expressed in 24-hour format. The date is expressed as mmddyy (Month Day Year). The word NIX, in this example, is the password on the login directory.

When logging into the system, typing errors, incorrect passwords, etc., may cause error messages to be displayed. Most are self-explanatory; for a detailed discussion, see the New User's Guide to EDITOR and RUNOFF.

DIRECTORY OPERATIONS

Changing the Working Directory

After logging in, the user's working directory is set to the login UFD by PRIMOS. The user can move to another directory in the PRIMOS tree structure (i.e., attach) with the ATTACH command. The format is:

ATTACH new-directory

new-directory is the pathname of the new working directory.

Note

If any of the directories in the pathname have passwords, the entire pathname must be enclosed in single quotes, as in:

```
A 'BEECH SECRET>BRANCH5'
```

To set the MFD of a disk as the working directory, the format is slightly different:

ATTACH '<volume>MFD mfd-password'

volume is either the literal volume name or the logical disk number, and mfd-password is the password of the MFD. A password is always required for a MFD.

Recovering from Errors While Attaching: If an error message is returned following an ATTACH command (for example, if a UFD is not found), the user remains attached to the previous working directory.

Creating New Directories

To organize tasks and work efficiently, it is often advantageous to create new sub-UFDs. These sub-UFDs can be created within UFDs or other sub-UFDs with the CREATE command. They can contain files and/or other sub-directories. The format is:

CREATE pathname

The pathname specifies the directory in which the sub-UFD is being created, as well as the name of the new directory.

Example:

CREATE <1>TOPS>MIDDLE>BOTTOM

The sub-UFD BOTTOM is created in the UFD MIDDLE, which in turn is found in the MFD TOPS.

Two files or sub-UFDs of the same name are not permitted in a directory. If this is inadvertently attempted, PRIMOS will return the message: ALREADY EXISTS.

Assigning Directory Passwords

Directories may be secured against unauthorized users by assigning passwords with the PASSWD command. There are two levels of passwords: owner and non-owner. If you give the owner password in an ATTACH command, you have owner status; if you give the non-owner password in an ATTACH command, you have non-owner status. Files can be given different access rights for owners and non-owners with the PROTEC command (see Controlling File Access).

The PASSWD command replaces any existing password(s) on the working directory with one or two new passwords, or assigns passwords to this directory if there are none. The format is:

PASSWD owner-password [non-owner-password]

The owner-password is specified first; the non-owner-password, if given, follows. If a non-owner password is not specified, the default

is null; then, any password (except the owner password) or none allows access to this directory as a non-owner.

Example:

```
OK, A DOUROS NIX
OK, PASSWD US THEM
```

The old password NIX is replaced by the owner password US, and the non-owner password THEM.

Deleting Directories

When directories are no longer needed they may be removed from the system to provide more room for other uses. The DELETE command can also delete empty subdirectories from a given directory. The format is:

```
DELETE pathname
```

Sub-UFDs that are not empty, i.e., that still contain files or subdirectories, cannot be deleted with this command. All entries in the directory must be deleted first. If an attempt is made to delete directories containing files, PRIMOS prints the message:

```
DIRECTORY NOT EMPTY
```

Examining Contents of a Directory

After logging in or attaching to a directory, the user can examine the contents of this directory with the LISTF command which generates a list of the files and sub-directories in the current directory. The format is:

```
LISTF
```

For example: the working directory is called LAURA; the following list will be generated when LISTF is entered at the terminal:

```
OK, LISTF

UFD=LAURA  6 OWNER

$QUERY  BOILER  EX       LETTER  QUERY   OLISTF  BASICPROGS
OUTLINE $OUTLINE         MQL     $MQL    $LETTER MQL.LETTER      FTN1Ø
EXAMPLES         FUTIL.1Ø         $FUTIL.1Ø

OK,
```

The number following the UFD-name is the logical device number, in this case, 6. The words OWNER or NONOWN follow this number, indicating the

user status in this directory. (See Securing Directories).

If there are no files contained in a directory, .NULL. is printed instead of a list of files.

SYSTEM INFORMATION

Table 2-2 summarizes useful information you may need about the system and how to obtain it.

Table 2-2
Useful System Information

Item	Used to	PRIMOS commands
Number of users	Indicate system resource usage and expected performance.	STATUS USERS (user list) USERS (number of users)
User login UFD	Identify user who spooled text file (printed on banner).	STATUS, STATUS UNITS
User number		STATUS
User line number	Change terminal characteristics.	STATUS
User physical device		STATUS
Open file units	Avoid conflict when using files.	STATUS, STATUS UNITS
Disks in operation		STATUS, STATUS DISKS
Assigned peripheral devices	Tell if spool printer is working; if devices are available.	STATUS USERS
User priorities		STATUS USERS
Other user numbers		STATUS USERS
Your phantom user number	Log out your phantoms.	STATUS USERS
Network information	Tell if network is available.	STATUS, STATUS NET
Login nodename		STATUS, STATUS UNITS
Records available	Tell if there is enough room for file building, sorting, etc.	AVAIL
System time and date	Perform time logging in audit files.	DATE
Computer time used since login	Measure program execution time.	TIME

Spool queue contents	Tell if job has been printed.	SPOOL -LIST
CX queue contents	Tell if job has been completed.	CX -ALL

Note

Any information given by any STATUS command is also given by the STATUS ALL command.

FILE OPERATIONS

Creating and Modifying Files

Text files may be created and modified using the text editor (ED). Files may be transferred from other systems using magnetic tape (MAGNET command), or paper tape (BASINP command). See Reference Guide, PRIMOS Commands for more details on these commands.

Changing File Names

It is often convenient or necessary to change the name of a file or a directory. This is done with the CNAME command. The format is:

```
CNAME old-name new-name
```

old-name is the pathname of the file to be renamed, and new-name is the new filename.

Example:

```
CN TOOLS>FORTRAN>TEST OLDTEST
```

The file named TEST in the sub-UFD FORTRAN in the UFD TOOLS is changed to OLDTEST. Since no disk was specified all MFDs (starting with logical disk Ø) are searched for the UFD TOOLS.

If new-name already exists, PRIMOS will display the message:

```
ALREADY EXISTS
```

An incorrect old-name prompts the message:

```
NOT FOUND
ER!
```

Determining File Size

The size (in decimal records) of a file is obtained with the SIZE command. This command returns the number of records in the file specified by the given pathname. The number of records in a file is defined as the total number of data words divided by 448. However, a zero-word length file always contains one record. The format is:

```
SIZE pathname
```

Example:

```
OK, SIZE GLOSSARY
```

```
GO
    14 RECORDS IN FILE

OK,
```

Examining File Contents

Contents of a program or any text file can be examined at the terminal with the SLIST command. The format is:

SLIST pathname

The file specified by the given pathname is displayed at the terminal.

Obtaining Copies of Files

Printed copies of files from a line printer are obtained with the SPOOL command. It has several options, some of which will not apply to all systems, as systems may be configured differently. The format is:

SPOOL pathname

PRIMOS makes a copy of pathname in the Spool Queue List for the line printer, and displays the message:

YOUR SPOOL FILE IS PRTxxx (length)

xxx is a 3-digit number which identifies the file in the Spool Queue List. The reason for a list, rather than just having each file spooled out as the request comes, is that some requests are very long - hundreds of pages. PRIMOS spools out the shorter files as soon as possible, rather than make the user wait while the long files are printed. The length (SHORT or LONG) which follows the SPOOL message is the category to which the file has been assigned. It is possible to check the status of a SPOOL request by giving the command:

SPOOL -LIST

Example:

```
OK, SPOOL $S2.3Ø57
GO
YOUR SPOOL FILE IS PRTØØ6  (LONG)  REV 15.2**

OK, SPOOL -LIST
GO

 USER    FILE    DATE/TIME     OPTS   SIZE  NAME          FORM     DEFER
------  ------  -----------  ------  ----  ----------  ------  -----
SOPHIE  PRTØØ5  1Ø/25 14:26  S            5  $UNFUNDED   W.WIBA
TEKMAN  PRTØØ6  1Ø/25 15:46  L           22  $S2.3Ø57
```

OK,

To cancel a spool request, the command format is:

SPOOL -CANCEL PRTxxx

xxx is the number of your Spool File.

For example:

OK, SPOOL -CANCEL PRTØ13
GO
PRTØ13 CANCELLED.

OK,

Deferring Printing: The -DEFER option tells the Spooler not to begin printing the indicated file until the system time matches the time specified with DEFER. This also permits you to enter SPOOL requests at your convenience, rather than waiting for the appropriate hour.

Specify the DEFER option by:

SPOOL filename -DEFER 'time'

The value for 'time' can be expressed either in 24-hour format (ØØ:ØØ = Midnight) or in 12-hour format followed by AM or PM (12:ØØ AM = Midnight). The format for 'time' is 'HH:MM', where HH is hours, ":" is any character, and MM is minutes. If you specify -DEFER but omit time you will get the prompt:

ENTER DEFERRED PRINT TIME:

If 'time' is not in the correct format, you will get the above prompt again, plus this informational message:

CORRECT FORMAT IS HH:MM AM/PM.

Printing on Special Forms: Line printers traditionally use one of two types of paper -- "wide" listing paper, on which most program listings appear, and 8-1/2x11-inch white paper, which is standard for memos and documentation. Computer rooms often stock a variety of special paper forms for special purposes, such as 5-copy sets, pre-printed forms (checks, orders, invoices), or odd sizes or colors of paper.

Request a specific form by:

SPOOL filename -FORM form-name

form-name is any six-character (or less) combination of letters. A list of available form names should be obtained from the system Administrator.

Deleting Files

When files or programs are no longer needed they may be removed from the system to provide more room for other uses. The DELETE command deletes files from the working directory. The format is:

DELETE pathname

Controlling File Access

Assigning passwords to directories allows users working in a directory to be classified as owners or non-owners, depending upon which password they use with the ATTACH command. Controlled access can be established for any file using the PROTEC command. This command sets the protection keys for users with owner and non-owner status in the directory (see Assigning Directory Passwords above). The format is:

PROTEC pathname [owner-rights] [non-owner-rights]

pathname	The name of the file to be protected.
owner-rights	A key specifying owner's access rights to file (original value=7).
non-owner-rights	A key specifying the non-owner's access rights (original value=Ø).

The values and meanings of the access keys are:

key	Rights
Ø	No access of any kind allowed
1	Read only
2	Write only
3	Read and Write
4	Delete and truncate
5	Delete, truncate and read
6	Delete, truncate and write
7	All access

Note

The default protection keys associated with any newly created file or UFD are: 7 Ø: the owner is given ALL rights and the non-owner is given none.

Example:

```
PROTEC <OLD>MYUFD>SECRET 7 1
```

In this example, protection rights are set on the file SECRET in the UFD MYUFD so that all rights are given the owner and only read rights are given the non-owner.

COMPLETING A WORK SESSION

When finished with a session at the terminal, give the LOGOUT command. The format is:

```
LOGOUT
```

PRIMOS acknowledges the command with the following message:

```
UFD-name (user-number) LOGGED OUT AT (time) (date)
TIME USED = terminal-time CPU-time I/O-time
```

user-number	The one assigned at LOGIN.
terminal-time	The amount of elapsed clock time between LOGIN and LOGOUT in hours and minutes.
CPU-time	Central Processing Unit time consumed in minutes and seconds.
I/O-time	The amount of input/output time used in minutes and seconds.

It is good practice to log out after every session. This closes all files and releases the PRIMOS process to another user. However, if you forget to log out, there is no serious harm done. The system will automatically log out an unused terminal after a time delay. This delay is set by the System Administrator (the default is 1000 minutes but most System Administrators will lower this value).

SECTION 3

USING BASIC/VM

INTRODUCTION

This section is an overview of all the basic concepts and commands you need to work with BASIC/VM. The first part discusses elementary tasks and the commands needed to do them; the second part expands on some of the features introduced in the first part.

ENTERING THE BASICV SUBSYSTEM

From PRIMOS command level, it is possible to access any of the subsystems available under PRIMOS. To enter the BASICV subsystem, type BASICV. The system then responds with the following message and prompt:

```
OK, BASICV
BASICV REV 16.Ø
NEW OR OLD:
```

The 'NEW or OLD:' prompt asks you to specify whether a new file is to be created or if an OLD, or previously created file is to be called from the current working directory to your working area in BASIC/VM. Type either OLD or NEW followed by the name of the file you wish to access or create. Remember that all command-line input must be in upper case only.

Calling an OLD File

The directory from which you gave the 'BASICV' command is your current working directory. It is referred to as 'the foreground' in BASIC/VM. A file that is currently open (and being edited, created or run) in this working directory is called the foreground file. Only one file can be in the foreground at any time.

When you type OLD followed by a filename (or pathname) in response to the first prompt, BASICV locates the file and makes it the foreground file. For example, if you want to call a file named JUNK to the foreground, type:

```
>OLD JUNK
>
```

The system responds with the right angle bracket, indicating that you are now at BASICV command level. This angle bracket is the compiler's prompt character. If a file or pathname is not specified after OLD, the system prompts for it:

```
NEW OR OLD: OLD
OLD FILE NAME: JUNK
>
```

If the file is in your current directory, a filename is sufficient; otherwise, a pathname (see Appendix D for definition) must be specified. The latter part of this section outlines how to access files in directories other than the current one.

Entering A New File

To enter a new program at the terminal, type NEW followed by a name for the file you wish to create. If a filename is not specified, BASICV will ask for one.

Example:

```
NEW OR OLD: NEW
NEW FILENAME: TEST
>
```

This new file becomes the foreground file. It remains in the foreground until an OLD file is called in or you decide to create another new file.

USING BASICV COMMANDS

Now that the preliminaries are out of the way, you can create a new program or work with an existing one. Programming in BASIC/VM involves several routine operations which every user needs to be familiar with. The following is a list of these operations and the BASIC/VM commands that can be used to accomplish them:

- Displaying contents of current working directory (CATALOG)
- Displaying contents of foreground file (LIST)
- Displaying contents of non-foreground file (TYPE)
- Saving a NEW or modified file (FILE)
- Running a foreground program (RUN)

- Checking for syntax errors (COMPILE)
- Translating source program into executable machine language (COMPILE)
- Executing a compiled source program (EXECUTE)
- Editing a file (simple techniques, e.g., deleting lines)
- Combining two or more programs (LOAD)
- Renaming a foreground file (RENAME)
- Removing files from a directory (PURGE)
- Exiting the BASICV subsystem (QUIT)

Examining Directory Contents: CATALOG

The BASIC/VM CATALOG command returns a list of all the files in the current working directory. It has several options which provide additional information about the files. The format of the command is:

CATALOG [options]

Where options are one or more of the following:

DATE	Date and time the file was last modified.
PROTECTION	Owner or non-owner protection attributes (see PRIMOS, Section 2).
SIZE	Number of records in each file.
TYPE	Describes file type (DAM, SAM, SEGSAM, SEGDAM, UFD; see Appendix D).
ALL	Gives all of the above option information.

If no options are specified, only the filenames are displayed. Option abbreviations are underlined.

Example:

>CATALOG A

	Size	Type	PASSWORD Owner	Nonowner	Time	Date
PRINTX	1	SAM	O:RWD	N:NIL	10:48:24	9/20/78
TAB	1	SAM	O:RWD	N:NIL	10:54:12	9/20/78
BASICPROGS		UFD	O:RWD	N:NIL	14:50:16	9/05/78
MAT	1	SAM	O:RWD	N:NIL	11:32:44	9/20/78
OUTLINE	3	SAM	O:RWD	N:NIL	16:54:36	8/25/78
$OUTLINE	3	SAM	O:RWD	N:NIL	14:20:36	8/07/78
AGES	1	SAM	O:RWD	N:NIL	11:41:00	9/12/78
MATREX	3	DAM	O:RWD	N:NIL	11:36:12	9/20/78
ACCUM	1	SAM	O:RWD	N:NIL	12:17:40	9/06/78
SECRET	1	SAM	O:RWD	N:R	9:41:52	9/22/78
OTHER	1	SAM	O:RD	N:NIL	9:40:40	9/22/78
COMPILEX	3	DAM	O:RWD	N:NIL	10:15:40	9/07/78
PERSONAL	1	SAM	O:RWD	N:NIL	11:17:56	9/12/78

Displaying Contents of Foreground File

The LIST and LISTNH commands display all or part of the foreground file at the terminal. LIST displays a program header including the program name, date and time; LISTNH omits the header. The format is:

```
LIST
LISTNH   [line-number-1,...line-number-n]
```

If the line number options are specified, only the indicated line numbers are displayed. If omitted, the entire program is displayed.

Displaying Non-Foreground Files

Non-foreground files can be listed at the terminal with the TYPE command. This is useful for comparing programs. A TYPEd file does not become the foreground file; its contents are merely displayed on the user terminal. To modify or run the file (if it is a program), use the OLD command to bring it to the foreground. The format of the TYPE command is:

```
TYPE pathname
```

where pathname is the pathname or filename of a non-foreground file.

The following example illustrates a situation where the file called XYZ is in the foreground and a list of the file AGES is needed.

Example:

```
          (XYZ is in the foreground)
>TYPE AGES
10 REM AGES
20 DATA 1952, 1956, 1957
30 READ Y1,Y2,Y3
35 INPUT 'ENTER THE CURRENT YEAR': Y
40 A1=Y-Y1
50 A2=Y-Y2
60 A3=Y-Y3
65 PRINT A1, A2, A3
70 END
>LIST                          (XYZ is still in foreground)
XYZ                TUE, SEP 12 1978          11:30:30
(etc.)
```

Saving a New or Modified File

A new file is entered into the system by typing the statements at the terminal in proper BASIC/VM form. Section 4 explains all statement syntax rules. All statements are preceded by line numbers to distinguish them from commands, like LIST and RUN, which are not preceded by line numbers. Use the error correction characters (" and ?), discussed in Section 2, to correct typing mistakes. When the entire program is entered, FILE it to ensure that a copy of it is saved for future use. If a new program is not FILEd, it will vanish when you leave the BASICV subsystem, or it will be overwritten when another file is called to the foreground. The FILE command writes a copy of the foreground file to disk under the name you specified in the NEW or OLD sequence. If you want to change the name of the file, simply specify a new name after the FILE command. The format is:

```
FILE [filename]
```

Once a NEW program has been filed, it becomes an OLD program. However, the file remains in the foreground until another replaces it.

Compiling the Source Code

In order for a program to be run or executed, the source code must be translated, or COMPILEd, into executable machine language. During the COMPILE process, the compiler parses the code for errors and produces a binary version of the source file that can be EXECUTed or RUN. This binary file is kept in user memory until the EXECUTE command is issued; it may optionally be named and saved to disk for future use by specifying a filename with COMPILE. The format of the COMPILE command is:

```
COMPILE [filename]
```

If a filename is specified, the binary version of the foreground source file will be stored on disk with the indicated name.

Checking for Syntax Errors

The COMPILE process also checks for syntax errors in a NEW or OLD foreground file. Syntax errors include misspelling of statements or referencing an undefined variable. During this process the compiler parses each line in the file and weeds out the errors. These errors are collectively referred to as 'compile-time' errors, as distinguished from 'run-time' errors which are displayed when a program is actually executed or run. The COMPILE process does not run the program; it translates the source code to binary form, looks for faulty lines, displays them, and indicates what is wrong with each one.

Most error messages are self-explanatory. For a complete list of error messages, see APPENDIX D. Errors uncovered by the COMPILE process can usually be corrected with simple edit procedures, discussed later in this section.

Executing a Program

After a program has been COMPILEd, or translated into binary form, it can be executed with the EXECUTE command. EXECUTE accepts a pathname option; therefore, it can run either a foreground or non-foreground file. The format is:

```
EXECUTE [pathname]
```

If a pathname is specified, EXECUTE will do one of two things: if the pathname is that of an executable binary, the program will be executed immediately; if the pathname is that of a non-compiled source program, the compiler first translates the code into executable form, then executes it.

When the EXECUTE command is given without the pathname option, the currently compiled code in user memory is executed. If no such executable code exists, the foreground file is compiled, then executed. Remember, an executable binary version (machine language) of any source program must exist before the program can be executed.

Run-time Errors

The EXECUTE process also performs the additional function of displaying errors that occur during run-time. Run-time errors include faults in program logic or control, such as a READ after a WRITE to a sequential file. Usually, run-time errors impair program execution and can often prevent a program from running at all. Each run-time error is displayed at the terminal as it occurs, e.g., as the compiler attempts to execute a faulty statement.

Should a program not run to completion, it should be examined for logic errors and corrected as necessary. The BASIC/VM debugging commands, discussed in Section 7, are often helpful in detecting and rectifying such program control errors.

RUNning a Program

Foreground source programs can also be executed with the RUN command. RUN combines both COMPILE and EXECUTE processes. It has no pathname option, and therefore can run only foreground programs. RUN translates source code to executable machine language and executes it immediately. No binary file can be stored via this process.

Unlike EXECUTE, RUN displays both compile-time and run-time errors, whereas EXECUTE displays only run-time errors. RUN also has a no-header option, NH which suppresses the program header (name, date, time, etc.) at run-time. Execution may be instructed to begin at a specified point in the program by specifying the appropriate line number. The format of the RUN command is:

```
RUN[NH] [statement-number]
```

Remember, only the foreground file can be run with this command.

Editing a File

There are some simple editing techniques useful in correcting the errors pointed out by COMPILE, EXECUTE and RUN. They include deleting lines, inserting new lines, and retyping lines. More advanced editing procedures are covered in Section 7. The procedures discussed here allow you to add, delete or replace statement lines without specific editing commands.

- To delete a specific statement line, type the line number followed by a carriage return (CR).
- To insert a new statement line anywhere in the program, type the appropriate line number, followed by statement text. The new line is automatically placed in correct numerical sequence.
- To replace a statement line, type the line number followed by new statement text. The new statement will overwrite the original.

After a file has been edited or modified, be sure to FILE it so that the changes will be made permanent.

Process of a BASIC/VM Program

The steps typically taken in the creation of a BASIC/VM program are outlined in Figure 3-1. Not all the options that can be chosen in program development are included in this flow chart. Only the most commonly exercised options are shown.

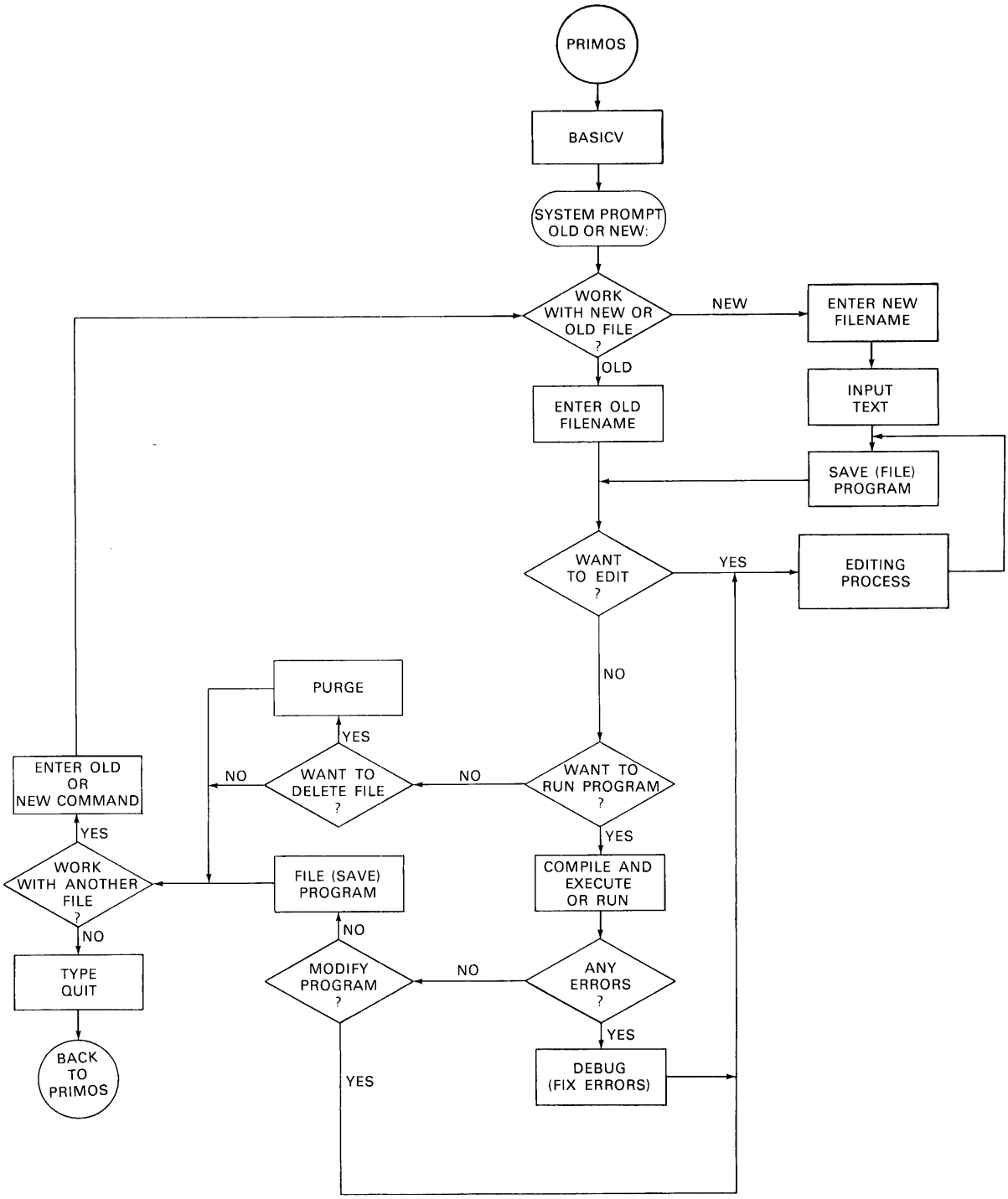


Figure 3-1. Process of a BASIC/VM Program

Sample Program

The following program demonstrates the use of simple editing techniques to correct errors displayed when the program is COMPILEd and EXECUTEd.

```
OK, BASICV                          (enter BASICV subsystem)
GO

BASICV REV16.2
NEW OR OLD: NEW SAMPLE              (create NEW file)

>10 DATA 12.1,34,78
>20 READ X,Y,Z,A
>30 PINT X,Y,Z
>40 END
>COMPILE                            (check for syntax errors)
30 PINT X,Y,Z
   ^
INVALID WORD IN STATEMENT
>30 PRINT X,Y,Z                     (correct misspelling of 'PRINT')
>COMPILE                            (no more syntax errors)
>EXECUTE
END OF DATA  AT LINE  20            (run-time error:  not
                                    enough data.  Program stops)

>15 DATA 10,20,30                   (insert new line to add more data)
>COMPILE                            (program now compiles and
                                    executes without errors)
>EXECUTE
12.1                34                          78
>FILE                               (program is o.k.;  FILE to save)
>QUIT                               (leave BASICV)

OK,                                 (back to PRIMOS!)
```

Combining Two or More Programs

The LOAD command may be used to combine two or more BASICV programs to form a single executable program. Usually, LOAD is employed when appending an external file to a foreground one. Line numbers which are common to both programs are overwritten in the foreground file by those line numbers in the external or LOADed program. Otherwise, line numbers are interwoven.

If a binary version of a program (compiled source code) is LOADed, it is stored in user memory but does not become part of the foreground file. If the EXECUTE command, with no pathname option, is issued susbsequent to this LOAD, the just-LOADed binary code will be executed.

The following examples illustrate two ways in which programs can be combined with LOAD.

Example 1 :

1. Source files:

```
GAME                          CALC
10 PRINT                      110 B=23
20 REM A MATH GAME            120 PRINT A,B
30 A=7                        .
.                             .
.                             .
.                             200 END
100 PRINT A*A

>OLD GAME                     ! GAME is now in foreground
>LOAD CALC
```

Foreground file now contains:

```
10 PRINT
.
.
.
100 PRINT A*A
110 B=23
120 PRINT A,B
.
.
.
200 END
```

Example 2:

2. Source files:

```
PROG                          PROG1
10 PRINT 'A'                  10 PRINT
20 A=1                        15 REM PROG1
30 B=2                        20 A=3
40 PRINT A*B                  30 B=5
                              35 C=11
                              45 PRINT A,B,C
                              55 END

>OLD PROG                     ! PROG is now in foreground
>LOAD PROG1
```

Foreground file now contains:

```
10 PRINT
15 REM PROG 1
20 A=3
30 B=5                        Statements 10, 20 and 30
35 C=11                       from PROG1 overwrite
40 PRINT A*B                  those in PROG.
45 PRINT A,B,C
55 END
```

Renaming a Foreground File

The name of a foreground file can be changed with the RENAME command. The format is:

RENAME new-filename

Only the new name of the file need be specified. Be aware that this command merely changes the name of the foreground file and does not change the name of the file on disk unless it is FILEd. When the renamed file is FILEd, two copies of the same file will exist: the original file and the renamed file.

Deleting a File from a Directory

To remove a file from a directory, use the PURGE command. The format is:

PURGE [pathname]

If no pathname is specified; the foreground file is deleted, otherwise the file indicated by pathname is removed.

EXITING THE BASICV SUBSYSTEM

There are two ways to exit the BASICV subsystem and return to PRIMOS command level. They are:

1. QUIT
2. CTRL-P, BREAK (see Section 2)

As explained in Section 2, CTRL-P and BREAK have the same overall effects. Both leave files open and can cause files to be truncated or otherwise garbled. It is advisable to type QUIT, which closes all open files, protecting them from unwanted modification. The QUIT command can only be used at command level, i.e., in response to the '>' prompt.

If one of the alternatives to QUIT is used, C ALL (short for CLOSE ALL) can be typed upon returning to PRIMOS command level. In most cases, this will ensure that all files are closed and that no damage will be done to any of them unintentionally.

Sometimes temporary files are left in the directory by the BREAK or CTRL-P methods. These are indicated by the characters T$ followed by four digits, T$nnnn, where nnnn ranges from ØØØØ to 9999. To check for these temporary files, type LISTF (at PRIMOS command level). To delete them, simply type DELETE followed by the T$ filename representation. For purposes of file integrity, it is recommended that BREAK and CTRL-P not be employed unless unavoidable (as in a long listing when the user is not at command level).

ADDITIONAL FEATURES

The following information deals with other features of BASIC/VM which may be of use to some programmers. These features include:

- Accessing files in directories other than the current working directory.
- Using the BASIC/VM ATTACH command to change the working directory.
- Running BASIC/VM programs from PRIMOS level.
- Modes of operation (or interpretation) in BASIC/VM.

ACCESSING FILES IN REMOTE DIRECTORIES

Accessing files in BASIC/VM is similar to the process previously described in Section 2. However, there are some important variations. The following examples illustrate file access procedures for all possible file location situations. User input is underlined.

Accessing a File in the Current UFD

To access a file called OLDF1 in the current UFD the format is:

```
NEW OR OLD: OLD OLDF1
```

Accessing a file in a sub-UFD in current UFD

To access a file called OLDF2 in a sub-UFD called SUBS, located in the current directory, the format is:

```
NEW OR OLD: OLD *>SUBS>OLDF2
```

The right angle brackets (>) indicate that the file is contained within the sub-UFD. The asterisk (*) means 'current UFD' and must be included in the pathname.

Accessing a File in a UFD on Another Disk

The general format of this procedure is:

```
OLD   <volume>
      <ldisk>
```

The parameter volume is the name of the disk on which the file is found.

For example, to access a file called FILEOK listed in a UFD called PROJECTS on a disk called SOFTWR, the following pathname would be typed:

```
OLD  <SOFTWR>PROJECTS>FILEOK
```

If the disk number was 3, the following format would be used:

```
OLD  <3>PROJECTS>FILEOK
```

If passwords are required on either the UFD or file, they are inserted after the directory-name requiring the password.

Example:

```
OLD <3>PROJECTS SECRET>FILEOK
```

In this case, the UFD PROJECTS is protected by the password SECRET.

ATTACHing to a Directory

The ATTACH command, discussed in Section 2, is used in BASIC/VM as well as in PRIMOS. However, like all BASICV commands, ATTACH cannot be abbreviated.

Attaching to a sub-UFD in the Current UFD

To ATTACH to a sub-UFD in the current UFD, the asterisk symbol is used to indicate the current directory. For example, to attach to a sub-UFD called PLAYS in the current directory, the format is:

```
ATTACH *>PLAYS
```

Attaching to a sub-UFD in Another UFD

When attaching to sub-UFD in a UFD other than the current directory, the UFD name, and password, if any, must be specified.

Example:

```
ATTACH NUMBERS>DECIMAL
```

The UFD-name is NUMBERS and the sub-UFD-name is DECIMAL.

Attaching to a UFD or sub-UFD on Another Disk

Attaching to a UFD or sub-UFD on another disk is the same as in PRIMOS. A volume name of the disk is given, or a logical disk number, (ldisk) is specified in the pathname. For example, to attach to a sub- UFD called PEONIES listed under a passworded UFD called FLOWERS on logical disk number 5, the format would be:

```
ATTACH <5> FLOWERS PASSWD>PEONIES
```

The password on FLOWERS is PASSWD and is required in the pathname.

RUNNING PROGRAMS FROM PRIMOS

A previously created and filed BASIC/VM program can be run from PRIMOS command level using the BASICV command. The format is:

```
OK,BASICV pathname
```

where pathname is either the source or binary form of a BASIC/VM program. The specified file is then run, leaving the user at PRIMOS command level.

Example:

The following is a BASIC/VM program, called 'T'.

```
1Ø PRINT 'SAMPLE'
2Ø FOR I=1 TO 1Ø
3Ø J=J+I
4Ø PRINT I, J
5Ø NEXT I
6Ø PRINT 'END OF LOOP'
7Ø PRINT 'YOU ARE NOW AT PRIMOS COMMAND LEVEL'
8Ø END
```

The program is now run from PRIMOS command level:

```
OK, BASICV T
GO
SAMPLE
1                    1
2                    3
3                    6
4                    1Ø
5                    15
6                    21
7                    28
8                    36
9                    45
1Ø                   55
END OF LOOP
YOU ARE NOW AT PRIMOS COMMAND LEVEL
OK,
```

MODES OF OPERATION IN BASIC/VM

There are three ways in which BASIC/VM can interpret terminal input:

- as a command (Command mode)
- as an executable statement (Immediate mode)
- as a line-numbered statement (Program-statement mode)

Any response issued in response to the BASICV prompt '>' is interpreted by the compiler as one of the above.

Command mode: If a command, e.g., RUN, is issued at this level, the compiler interprets it as a command and executes it immediately. Commands are directives to the system. More details appear in Section 4.

Program-statement mode: Statements entered with preceding line numbers are immediately stored in user memory as part of a program and are not compiled or executed until the compiler is instructed to do so. More details on program composition are found in Section 4.

Immediate mode: If a statement is entered without a line number, the compiler checks to see if it is executable, then attempts to execute it. Any errors in statement syntax are displayed immediately. These statements are not stored in user memory and cannot be stored as part of a program.

Immediate mode is useful for debugging programs, displaying local variable contents and for performing quick calculations. Immediate mode is also referred to as 'desk-calculator' mode.

When performing calculations, values for defined variables are placed in temporary storage locations; they remain intact until the variable is changed by some subsequent arithmetic operation. To clear all storage locations, and thereby reset all strings and numeric variables to null or zero values respectively, use the CLEAR command.

The following is an example of an Immediate mode terminal session. User input is underlined for clarity.

Example:

```
>A=12                          (performing simple calculations)
>B=25
>C=A+B
>PRINT C
37
>DIM A(3)                      (define matrix)
>A(1)=13
>A(2)=45                       (assign values to matrix elements)
>A(3)=56
>PRINT A
12                             (variable A and matrix A have the same
                                  name but different values)
>MAT PRINT A
13                  45                      56

>CLEAR                        (clear all variable storage locations)
>PRINT A
Ø
>PRINT B                      (all variables are now zeroed)
Ø
>PRINT C
Ø
>MAT PRINT A
UNDEFINED MATRIX AT LINE Ø    (no matrix definition exists
                                  after CLEAR)
>QUIT
```

SECTION 4

ELEMENTS OF BASIC/VM

LANGUAGE ELEMENTS

BASIC/VM is composed of the following elements:

- COMMANDS, which give directions to the system
- STATEMENTS, which make up programs
- EXPRESSIONS, which are combinations of operators and operands:
- OPERANDS, which are data elements including:

  constants
  variables
  functions
  arrays
  matrices

- OPERATORS, of four types, which manipulate operands:

  arithmetic
  logical
  relational
  string

The above elements are defined briefly in this section. Additional information on each may be found in other sections of this manual as indicated.

BASIC/VM uses the full ASCII character set including:

1. Letters from A-Z
2. Digits from Ø-9
3. Special Characters (see list in Appendix B)

OPERANDS

Within the context of a program, constants, functions, (or function references), variables and arrays are referred to as operands because they are operated on or manipulated by operators. Operators, like addition (+) or subtraction (-), tell a program what to do with the operands they connect.

Constants

A constant can be either a number or a quoted literal string. Its value does not change during the execution of a program.

Numeric Constants: are positive or negative integers, decimal or exponential expressions. BASIC/VM supports double-precision, floating-point numeric data, with a level of accuracy to 13 significant figures in the mantissa and 2 significant figures in the exponent.

Examples:

8.88

-123

2.5E-2

Literal String Constants: are a sequence of characters enclosed within single quotes or apostrophes ('). All spaces enclosed in the quotes are included in the string value. The maximum length of a string constant is 16Ø characters. See Section 12.

Examples:

'X,Y,Z'

'' (a null string)

'BASIC IS FUN!'

'VOITCH'

Variables

Variables are representations of data to which values are assigned. BASIC/VM supports both numeric and string scalar variables, and numeric and string subscripted variables, also known as arrays. Table 4-1 gives examples of legal and illegal variables in BASIC/VM.

Numeric Scalar Variables: are single letters (A-Z) optionally followed by a single digit (Ø-9). There are 286 possible combinations. Numeric scalar variables, also called simple numeric variables, are initialized by the BASICV subsystem to Ø at the start of the program in which they are defined. See Section 11.

Examples:

A2 Initial Value: A2=Ø

X4 Initial Value: X4=Ø

String Scalar Variables: are single letters (A-Z) followed by a decimal digit and a dollar sign($) or by a dollar sign alone. String scalar variables, also called simple string variables, represent character strings of various lengths and are initialized to null at the beginning of the program in which they are defined. See Section 12.

Examples:

B$

A2$

Numeric Subscripted Variables (arrays): are simple variables followed by one or two values enclosed in parentheses. An array name is a simple numeric variable, e.g., A. A single subscripted variable, e.g., A(1), refers to an element in a one-dimensional array named by A. A variable with two subscripts, e.g., A(1,2) references an element in a two dimensional array. Arrays and matrices are defined by the DIM statement or a MAT statement. See Section 9 for details.

String Subscripted Variables (arrays): are simple string variables followed by one or two values enclosed in parentheses. String arrays or matrices are named by simple string variables, and are dimensioned with a DIM or MAT statement. String array elements are represented by singly or doubly subscripted string variables. See Section 9 for details.

Examples:

A(4) one-dimensional numeric array element

A$(3,4) two-dimensional string array element

Table 4-1. Legal and Illegal Variables

Variable Type	Legal		Illegal	
numeric scalar	A2 X4	A Z	AB1 X14	AR BZ
string scalar	B$ A2$		AB$ A21$	AB3$
numeric subscripted	A2(1) A(1)	A(1,2) A2(1,2)	A12(1) AB(1,2)	A(1,2,1)
string subscripted	A$(1) A2$(4)	A$(1,2) A2$(1,2)	A12$(1,2) AB$(1)	

Functions

BASIC/VM provides a variety of numeric and string system functions e.g., TAN, COS, LEN, to operate on numeric and string data. Users may also define their own numeric and string functions, known as user-defined functions. See Section 1Ø for details.

Numeric Functions: are identified by a three or four letter name, followed by parenthetically enclosed numeric items or parameters. They specify an operation to be performed upon the numeric items to produce a single value. Table 1Ø-2 lists all available numeric functions. User-defined numeric functions are identified by the letters FN, followed by a simple numeric variable. (e.g., FNA, FNA8).

String Functions: are identified by a three to five letter name, followed by parenthetically enclosed string or numeric parameters. String functions are used to return information about strings and portions of strings, or to convert a numeric item to its corresponding string representation, and vice-versa. All string system functions are listed in Table 1Ø-2. User-defined string functions are named by the letter FN followed by a simple string variable, e.g., FNQ$.

OPERATORS

Within a program, the previously defined operands are combined with operators to form expressions. Operators specify what is to be done with the operands in these expressions, i.e., how they are to be evaluated. There are four types of operators: arithmetic, relational, logical and string. Operands, then, can be manipulated arithmetically, logically, relationally, or by string operators.

Arithmetic Operators

Arithmetic operators are of two types, unary or binary. Unary operators require only one operand, e.g., +7. They indicate the sign of the number. Binary operators require at least two operands, e.g., A1+7. The following table lists the arithmetic operators for BASIC/VM.

Arithmetic Operators

OPERATOR	DEFINITION	EXAMPLE
+	addition (unary positive)	A+B, +A
-	subtraction (unary negative)	A-B, -A
*	multiply	A*B
/	divide	A/B
^ or **	exponentiation	A^B,A**B
MOD	remainder from division modulus	A MOD B
MIN	select lesser value	A MIN B
MAX	select greater value	A MAX B

Relational Operators

Relational operators are used with conditional statements and statement modifiers. (See Section 6 for details.) There are six relational operators, as listed in the following table:

Relational Operators

OPERATORS	MEANING	EXAMPLE
<	less than	A<B
>	greater than	A>B
=	equal	A$=B$
<= or =<	less than or equal	A$<=C$
>= or =>	greater than or equal	A=>C
<> or ><	not equal	A<>D

String Operators

String operators include the above relational operators plus a concatenation operator (+) for combining two strings. String operands can only be used with string operators.

Logical Operators

Logical operators are connectives for relational expressions, allowing the testing of many relations at once.

Logical Operators

OPERATOR	MEANING	EXAMPLE
AND	true if both A and B are true	A AND B
OR	true if either A, B or both are true	A OR B
NOT	true if A is false	NOT A

EXPRESSIONS

Definition

Expressions are ordered combinations of operands and operators or other expressions which are evaluated within a program to produce a single result. Expressions can be arithmetic, e.g., A + B, relational, e.g., A>=B, or logical, e.g., A AND B.

Evaluation

Expressions are evaluated in order of operational priority. The priority list from highest to lowest for BASIC/VM is listed below. Within each level the evaluation order is from left to right.

( ) Parenthetical Expressions

System and User-defined Functions

^ (or **) Exponentiation

NOT, Unary (+ -)

*, /, MOD

+, -

MIN, MAX

Relationals (=, >, <, =>, <=, <>)

AND

OR

COMMANDS AND STATEMENTS

Commands

BASIC/VM system commands are directives to the BASICV subsystem to perform some immediate function. They are distinguished from statements in that they are not preceded by a line number. A list of all BASIC/VM commands and statements appears at the end of this section. Some commands have optional parameters or arguments to further define the operation which they perform. A complete list of BASIC/VM command formats and definitions can be found in Section 14.

Statements

Statements are part of a program, and as such, are preceded by a line number. When they appear without line numbers they are executed immediately.

Example:

```
PRINT 12*154
1848
```

This 'desk calculator' utilization is called Immediate Mode.

Statement Syntax

Statements must adhere to the following rules:

1. Each statement must be entered in upper case letters.
2. Each statement must be contained on one line.
3. Statements must not exceed 16Ø ASCII characters in length.
4. Portions of the statement (i.e., string literals) which the user wishes processed verbatim must be enclosed in single quotes.
5. Statements cannot be abbreviated.

Statements should be separated from their identifying line numbers with a blank space to avoid misinterpretations.

Statement Numbers

Statement numbers are one to five digit integers ranging from 1 to 99999. Successive statements are generally numbered in ascending order. It is recommended that statements be numbered in increments of ten, e.g., 1Ø, 2Ø, 3Ø... A statement may be added between lines 1Ø and 2Ø, for example, without changing the other statements. Given the following program:

```
1Ø PRINT 'NAME'
2Ø PRINT 'ADDRESS'
3Ø PRINT 'CITY'
>RUNNH
NAME
ADDRESS
CITY
```

To insert a line between lines 10 and 20 and another between 20 and 30, add PRINT statements as indicated:

```
>15 PRINT
>25 PRINT
>RUNNH
NAME

ADDRESS

CITY
```

Comments

Programs may contain comments or remarks which serve as explanatory notes for the benefit of the reader. They are preceeded by the letters REM or by the exclamation mark, (!). Comments may appear by themselves on separate lines or may be appended to a line with the exclamation mark. Comments may contain lower-case characters.

Example:

```
10 REM THIS IS A REMARK

20 !  THIS IS A REMARK ALSO

30 X=1 !  set x equal to one
```

LIST OF COMMANDS AND STATEMENTS

The following table lists all available BASIC/VM system commands and language statements with a brief description of their functionality. Also included are references to other sections in this guide where more information on each command and statement may be found.

Table 4-2
List of Commands and Statements

Command	Description	Section
ALTER	Allows editing of a single line in a program using subcommands, listed in Table 14-1.	7,14
ATTACH	Attaches to a directory in BASICV subsystem specified by pathname (similar to PRIMOS ATTACH but works in BASIC environment).	3,14
BREAK {ON / OFF}	Sets and unsets breakpoints at specified line numbers for debugging. Maximum of 1Ø break points may be set. See LBPS.	7,14
CATALOG	Lists all filenames under current UFD; options return other file-related information.	3,14
CLEAR	Resets all previous numeric values to Ø, or string values to null, deallocates previously defined arrays and closes all open files.	5,14
COMINP	Calls a specified command file to foreground; reads and executes commands until a COMINP TTY is reached.	6,14
COMPILE	Translates a source file into executable binary form; displays syntax errors.	3,14
CONTINUE	Resumes program execution after a breakpoint or PAUSE.	7,14
DELETE	Deletes specified statement lines.	7,14
EXECUTE	Executes indicated program (binary or source); displays run-time errors.	3,14
EXTRACT	Deletes all except specified lines.	7,14
FILE	Saves all input and modifications to current (foreground) a specified program file; writes file to disk.	3,14
LBPS	Lists currently set breakpoints.	7,14

LIST[NH]	Displays the contents of current file at terminal. NH option suppresses header or program title.	3,14
LOAD	Merges or adds an external program to current (foreground) program.	3,14
NEW	Indicates to compiler that a new file is to be created in foreground. New filename must be specified.	3,14
OLD	Calls pre-existing file to current working area (foreground). Makes it current file.	3,14
PURGE	Deletes specified file from UFD; file must be closed.	3,14,15
QUIT	Returns control to PRIMOS from BASICV command level.	3,14
RENAME	Changes name of foreground file.	3,14
RESEQUENCE	Renumbers statement in current program.	7,14
RUN[NH]	Initiates compile and execute processes on current source program.	3,14
TRACE {ON / OFF}	Tests program logic; line numbers are displayed as corresponding statements are executed.	7,14
TYPE	Displays contents of specified non-foreground file at terminal. TYPE program does not replace program currently in the foreground.	3,14

Statement	Description	Section
ADD #	Adds record to MIDAS file opened on specified file unit.	8,15
CHAIN	Transfers program control to specified external program.	6,15
CHANGE	Converts ASCII character string to one-dimensional numeric array or vice-versa.	1Ø,15 APPB

CLOSE #	Closes file on specified unit(s).	8,15
COMINP	Stops execution of current program: calls specified command file to foreground.	6,14,15
DATA	Contains numeric and string constants to be accessed by READ statement.	5,15
DEFINE FILE #	Opens a file of specified type on indicated unit number with optional access restrictions (APPEND, READ, SCRATCH).	8,15
DEF FN	Without FNEND, defines one-line function with numeric or string scalar variable arguments; with FNEND, defines multi-line function.	1Ø,15
DIM	Defines dimensions of numeric or string array or matrix.	9,15
DO...DOEND	Defines a set of statements to be executed in association with IF-THEN statement pair.	6,15
ELSE DO	Optional alternative to DO-DOEND statement set.	6,15
END	Terminates program execution; no message is displayed.	6,15
ENTER[#]	A timed input statement: with # option, returns user-number assigned at LOGIN; also sets time-limit on input.	5,15
FOR	Defines beginning and end of loop and loop index; used optionally with STEP, WHILE, UNTIL, NEXT.	6,15
GOSUB	Transfers program control to internal subroutine: used with RETURN.	6,15
GOTO	Transfers program control to specified line; can be used conditionally with IF or ON.	6,15
IF	Makes executable statements conditional; can be used with GOTO, THEN, ELSE DO, etc.	6,15

INPUT[LINE]	Requests data to be entered from terminal. LINE option accepts entire line, including commas and colons, as one entry.	5,15
LET	Assignment statement: optional.	5,15
MARGIN (OFF)	Changes width of output lines. OFF option turns off margin checking.	5,15
MAT {CON / IDN / NULL / ZER}	Sets initial value of matrix elements to zero, identity, null or one.	9,15
MAT {* / + / -}	Performs addition, subtraction or multiplication on two matrices.	9,15
MAT {INV / TRN}	Calculates INVERSE or TRANSPOSE values of one matrix and assigns them to another.	9,15
MAT INPUT	Reads data from a terminal and assigns values to elements of specified matrix or matrices.	9,15
MAT PRINT	Prints an entire matrix (or matrices) at terminal.	9,15
MAT READ	Reads values from data list(s): assigns them to elements of matrix or matrices.	9,15
MAT READ [*] #	Reads data from external file and assigns them to a specified matrix or matrices. Optional * forces all data in current record to be read before new one is read.	8,9,15
MAT WRITE #	Writes an entire matrix (or matrices) to a file on specified unit.	8,9,15
NEXT	Defines end of loop begun by a FOR statement.	6,15
ON {GOSUB / GOTO}	Transfers program control to a subroutine (GOSUB) or to one of a list of statement numbers (GOTO).	6,15

ON END # GOTO	Establishes a line number to which program control will transfer when an END OF FILE occurs in disk file opened on specified unit.	6,15
ON ERROR [#] GOTO	Defines statement line to which program control will transfer when a run-time error occurs, in disk file, if # specified.	6,8,15
PAUSE	Suspends program process at line where PAUSE occurs; to resume, type CONTINUE (command).	7,15
POSITION #	For DA disk files; positions internal record pointer to a specified record.	8,15
PRINT	Can be used with LIN,TAB,SPA options to print formatted data at terminal.	5,14,15
PRINT USING	Prints data output formatted according to format strings. See Table 15-2.	5,15
READ	Reads numeric or string values from a DATA statement in a program.	8,15
READ [KEY] #	Reads data associated with record key in MIDAS file opened on indicated unit.	8,15
READ LINE #	Reads from a disk file an entire line of text, including commas, colons, as one data item.	8,15
READ [*] #	Reads from current record in file open on specified unit; pointer then positions to next sequential record. * allows one record to be finished before a new one is read. (Holds pointer at current record after READ is complete.)	8,15
REM	Indicates a remark to the user only.	4,15
REMOVE #	Removes specific key from MIDAS file; if primary key, removes associated data also.	8,15
REPLACE #	Deletes data files referenced by a segment directory. Moves pointer from deleted file to another file; zeroes old pointer.	8, 15

RESTORE [# / $]	Reuses list of data items beginning with first item in lowest numbered DATA statement; # option reuses numeric items; $ option reuses string items only.	5,15
RETURN	Returns control from subroutine to statement following GOSUB statement.	5,15
REWIND #	Repositions record pointer to top of DA or MIDAS file open on specified unit.	8,15
WRITE #	Writes data to current record of disk file opened on specified unit.	8,15
WRITE USING #	Generates formatted output including tabs, spaces and column headings and writes output to ASCII disk file file opened on specified unit.	8,15

SECTION 5

DATA INPUT/OUTPUT

INTRODUCTION

This section covers data exchange between programs and terminals, including various methods of formatting data output. The first part of this section deals with data input or the process of providing data to a program either from within a program or from the terminal. The statements involved in data input are:

- LET — Assigns values to variables.
- DATA — Provides data values for associated READ statement.
- READ — Reads values from DATA statement into a list of variables.
- RESTORE — Tells program to reuse data values from previous DATA statement.
- INPUT — Requests user input from terminal.
- INPUT LINE — Accepts entire line of terminal input as one datum.
- ENTER[#] — Timed input statement; # option reads user login number into an indicated variable.

The second part deals with data output, including the output of data from a program to a terminal or other output device. Data output involves the following statements:

- PRINT — Prints data values verbatim or prints values associated with specified variable.
- PRINT {TAB / LIN / SPA} — Prints data with spacing conventions (tabs, blank lines, etc.) dictated by modifiers.
- PRINT USING — Prints data according to format indicated by special format characters.

- MARGIN — Alters data output line length by increasing or decreasing right margin from the default (80 characters positions).

Data exchange involving the transfer of information between a program and external data files is covered in Section 8, FILE HANDLING.

DATA INPUT STATEMENTS

Assignment Statement

The LET statement can be used to preface statements that assign values to variables and array elements; however, the use of LET is optional in BASIC/VM; it is not essential to the assignment process. The statements 'LET A=5' and 'A=5' are equivalent.

Reading Data Lists

The READ and DATA statements are used when all data values are known in advance and can be included directly in the program text. READ and DATA must always be used together. The READ statement lists numeric or string variables, separated by commas. The DATA statement contains values which correspond to the type (numeric or string) and number of variables listed by READ. If the items in a list exceed the length of one line, they may be continued in subsequent READ or DATA statements.

Example:

```
10 DATA 5,10,15,10
20 DATA 2, 7.2
30 READ A1, A2, A3,B,C,X
40 PRINT 'PARTIAL SUM =' ; A1+A2+A3
45 PRINT 'PARTIAL SUM=';B+C+X
50 PRINT 'AVERAGE=';(A1+A2+A3+B+C)/X
60 END
>RUNNH
PARTIAL SUM =30
PARTIAL SUM=29.2
AVERAGE=7.222222222222
```

If there are more variables in the READ statement than items in the DATA statement, an END OF DATA AT LINE nnnn message, where nnnn is a program line number, will appear at run-time.

Example:

```
1Ø READ X,Y,Z,V,B
2Ø DATA 12,67,89
3Ø R=X+Y+Z+B
4Ø PRINT R
>RUNNH
END OF DATA AT LINE 1Ø
```

If the DATA statement contains more elements than there are variables in the READ statement, the extra values are ignored.

Recycling Data Values

The RESTORE statement enables recycling of data values within a program without the need to re-enter them. The subsequent READ statement is directed to reuse the data beginning with the first value in the lowest numbered DATA statement.

Example:

```
 5  READ X
10  PRINT 'LOOP = ':X:' TIMES'
20  PRINT 'FIRST VALUES OF Y ARE:'
25  X=X-1
30  FOR A = 1 TO X
40  READ Y
50  PRINT Y
60  NEXT A
70  RESTORE
80  PRINT 'SECOND VALUES OF Y ARE:'
90  FOR A = 1 TO X
100 READ Y
110 PRINT Y
120 NEXT A
130 DATA 5, 1, 3, 5, 7
140 END
```

This program yields two different sets of values for Y, one for each time the variable is passed through the loop. When run, the program produces the following output:

```
LOOP = 5 TIMES
FIRST VALUES OF Y ARE:
1
3
5
7
SECOND VALUES OF Y ARE:
5
1
3
5
```

It is also possible to RESTORE only numeric or string values, using the alternate forms of the RESTORE statement.

```
RESTORE # reuses all numeric items, and

RESTORE $ reuses all string items, beginning with
          lowest-numbered DATA statement.
```

Data Input From Terminal

The INPUT statement accepts data values from the user terminal. Multiple variables, both numeric and string, can be specified on the INPUT statement line. Values for each variable are then entered from the terminal at run-time. This feature allows data values to be varied each time the program is run, providing increased program flexibility. The default prompt is the exclamation point (!), which indicates that the program is awaiting terminal input.

Example:

```
10  REM AVERAGE ANY 3 NUMBERS
20  PRINT 'GIVE ME 3 NUMBERS'
30  INPUT A, B, C
40  X = (A+B+C)/3
50  PRINT 'THE AVERAGE IS:'  :X
60  END

GIVE ME 3 NUMBERS
!  10, 30, 50
THE AVERAGE IS:  30
```

The program asks for the input of any three numbers separated by commas. If the commas are omitted, the line is accepted as one entry and a second exclamation mark is printed indicating a second value is expected. If more items are entered than required, the extraneous ones are ignored.

Since the default INPUT prompt (!) is not very informative, an alternate form of the statement allows the inclusion of a prompt string. The format is:

```
INPUT ['prompt-string',] var-1[,...var-n]
```

Example:

```
10 INPUT 'INPUT YOUR FAVORITE COLOR', C$
20 PRINT C$:  'IS THE ICKIEST COLOR I EVER SAW!'
```

Of course, some sort of prompt statement may be printed prior to the INPUT statement, as in the first example above.

If commas must be included as part of your input, use the INPUT LINE statement which accepts an entire line, including commas, as one entry.

```
10  PRINT 'WHAT IS YOUR NAME AND ADDRESS'
20  INPUT LINE C$
30  PRINT
40  PRINT C$:  '-THANK YOU'
50  END
```

A sample output might be:

```
WHAT IS YOUR NAME AND ADDRESS

 (user inputs info in response to !  prompt)

!LOU PIAZZA, CLOUD NINE

LOU PIAZZA, CLOUD NINE -THANK YOU
```

Timed Terminal Input

The ENTER statement serves the same purpose as INPUT but allows the user to specify a time limit on response requested from the terminal, as well as a variable to indicate the actual time used. The ENTER statement has no prompt, and therefore it is helpful to include a prompt of some sort prior to the ENTER statement itself. The format of the ENTER statement is:

ENTER time-limit, time-limit-variable, { numeric-variable / string-variable }

where the time-limit is expressed in seconds, time-limit-variable represents the actual time the user needed to respond, and the numeric-variable or string-variable is the variable for which a value is expected from the terminal.

In the following example, a value for Z is expected from the terminal and a limit of 5 seconds is placed on response time. T represents the time-limit-variable.

Example:

```
10 PRINT 'Enter a value for Z'
20 ENTER 5,T,Z
30 PRINT T
40 PRINT Z
>RUNNH
Enter a value for Z
22
3
22
STOP AT LINE 40
```

The example shows that an input value of 22 was assigned to Z, and that the value was input in 3 seconds (T). If the time limit had been exceeded, the variable Z would have been set equal to zero and T would have been reported as -5.

Another form of ENTER, ENTER #, gets the user's login number (assigned at LOGIN time) and places it in a user-specified variable. The format is:

ENTER # user-num-var [,time-limit,time-limit-var,{num-var / str-var}]

where user-num-var is the numeric variable which represents the user number. Other options are the same as for ENTER (above).

Example:

```
>ENTER # U
>PRINT U
>28
```

The user's login number is 28 and is returned by printing the value of U. If a user-variable is not specified with ENTER #, an error will be displayed.

Example:

```
1Ø ENTER # T,5,H,P
2Ø PRINT 'YOUR USER NUMBER IS:':T
3Ø PRINT 'P=':P
4Ø PRINT 'YOUR RESPONSE TIME IN SECONDS WAS:':H
5Ø STOP
>RUNNH
!12
YOUR USER NUMBER IS: 28
P=12
YOUR RESPONSE TIME IN SECONDS WAS: 2
STOP AT LINE 5Ø
```

DATA OUTPUT STATEMENTS

The results of data manipulations performed within a program are not displayed at the terminal unless some form of the PRINT statement is included in the program. The remainder of this section describes various methods of printing and formatting data using the PRINT statement and the formatting modifiers, LIN, SPA, TAB, and MARGIN.

Default PRINTing

Without the use of formatting modifiers, the PRINT statement can accomplish the following simple formatting tasks:

1. Inserting blank lines in the output.

2. Separating data into columns (using commas).

3. Spacing data items on a line (using colons or semicolons).

4. Conditionally printing a line (with WHILE, UNTIL, etc.).

The following program demonstrates the simple formatting of string and numeric values in two columns by using commas:

```
10 PRINT 'MATH CALCULATIONS'
20 PRINT
30 PRINT 'ADD', 'MULTIPLY'
40 A=3
50 B=7
60 PRINT
70 PRINT A+B, A*B
80 END
>RUNNH
MATH CALCULATIONS

ADD                   MULTIPLY

10                    21
```

The PRINT statements on lines 20 and 60 each cause a single blank line in the output. Enclosing a string in single quotes in a PRINT statement causes the string to be printed verbatim. Separating more than one item with commas causes each item to be printed in a separate column.

Column Separators: The output from the PRINT statement is normally divided into zones or columns of 21 characters each. The first zone starts in column 1, the second in column 22, etc. For the average printing page, the maximum number of zones is five.

A comma in a print list causes the terminal to advance to the first character position of the next available zone. If line overflow occurs, the current line is printed and a new line is started. If the last element of the print list is a comma, the partial line, if any, is printed and the cursor is positioned to the start of the next available zone.

Example:

```
10 PRINT 'COL.1', 'COL.22', 'COL.43'
20 PRINT
30 A$ = 'NAME'
40 B$ = 'ADDRESS'
50 C$ = 'PHONE NO.'
60 PRINT A$, B$, C$
65 END
```

When run, the following output results:

```
COL.1                COL.22               COL.43

NAME                 ADDRESS              PHONE NO.
```

Spacing Items: Using a colon (:) in a PRINT statement causes output items to be separated by a single space. Using a semicolon (;) causes no characters to be placed between output items. The following example shows how a phrase can be output in at least three different ways by using commas, semi-colons and colons in PRINT statements.

Example:

```
10 A$='COTTON'
20 B$='CANDY'
30 C$='IS'
40 D$='STICKY'
50 PRINT A$,B$,C$,D$
55 PRINT
60 PRINT A$;B$;C$;D$
70 PRINT
80 PRINT A$:B$:C$:D$
85 PRINT
90 END
>RUNNH
COTTON          CANDY           IS              STICKY

COTTONCANDYISSTICKY

COTTON CANDY IS STICKY
```

PRINTing Data With Modifiers

In addition to the delimiters discussed above, the PRINT statement also takes three optional modifiers. They format output by forcing items to indicated tab positions, by inserting any number of spaces between items, and by inserting any number of blank lines between lines of output.

TAB modifier: A specific tab position may be indicated by the TAB modifier followed by a parenthetical value representing the column number.

Example:

```
10 PRINT'COL.1';TAB(40);'COL.40'
20 PRINT
30 X=3^2
40 Y=X*50
50 PRINT X;TAB(40);Y
60 END
>RUNNH
COL.1                                    COL.40
9                                        450
```

SPA modifier: A specific number of spaces may be forced between items in the output by the SPA modifier followed by a parenthetical value representing the number of blank character positions.

Example:

```
10 PRINT SPA(5): 'COL.5'
20 PRINT
30 X=5
40 Y=X*5
50 PRINT X;SPA(5);Y
60 END
>RUNNH
     COL.5

5     25
```

LIN Modifier: A specific number of blank lines may be forced between items in one PRINT statement using the LIN modifier followed by a parenthetical value. This eliminates the need for consecutive PRINT statements.

Example:

```
10 PRINT 'COL.1'
20 PRINT
30 X=3^2
40 Y=X^2
50 PRINT X;LIN(3);Y
55 END
>RUNNH
COL.1

 9


 81
```

Note

LIN(3) outputs three Carriage Return - Line Feed combinations. LIN(-3) outputs three Line Feeds without Carriage Returns. LIN(Ø) outputs a carriage return without a Line Feed.

Formatting With PRINT USING

Additional formatting capabilities are provided by the PRINT USING statement in conjunction with a series of format characters. Both numeric and string data output can be formatted according to a field of special format characters, called a format string. This format string is included on the PRINT USING statement line prior to the list of items to be printed. The field may contain either numeric or string format characters, depending on the type of data to be formatted.

There are seven special characters which define numeric format:

. , ^ + - $

Table 15-2 in the reference section lists each character and several examples of its use.

The # Sign: One or more pound signs (#) in a format string represent digit positions which will be occupied by the datum to be formatted. which is filled with the data provided in the statement.

Example:

```
PRINT USING '##', 25
```

Results in the output: 25

Including too few pound signs for a non-decimal datum causes a row of asterisks to be printed. This indicates that the item to be formatted was too large for the specified field. For example:

```
>PRINT USING '####', 123456
****
```

The period (.): represents the position at which a decimal point should occur in the datum to be printed:

```
PRINT USING '##.##', 20
```

Results in the output: 20.00

(Digit positions to the right of the decimal point will be filled with zeroes.)

Note

If too few digit places are specified in the format for a decimal number, the item will be rounded off as follows:

40.325 = 40.32
40.327 = 40.33
40.323 = 40.32

The comma (,): represents a comma in the corresponding position of the output unless all digits prior to the comma are zero. In that case, a space is printed in the corresponding comma position.

Examples:

```
PRINT USING '#,###.##', 2000
```

Results in the output: 2,000.00

```
PRINT USING '+#,###.##', 030.6
```

Results in the output: + 30.6

The up arrow (^): is used in sets of four. Four up arrows indicate an exponent field which will be output as E+nn where nn is a two-digit number depending on how many places the decimal is moved in the format.

Example:

```
PRINT USING '####^^^^', 17.35
```

Results in the output: 1735E-Ø2

Plus and minus signs (+ -): are used to indicate the value of a datum to be printed. A single plus sign placed in either the first or last character position of the format causes either a + or - sign to be printed in front of the item, depending on whether it is positive or negative.

Examples:

```
PRINT USING '+##.##', 25
```

Results in the output: +25.ØØ

```
PRINT USING '+##.##', -12.3
```

Results in the output: -12.3Ø
how about 'PRINT USING '##.##-' -32

Two or more plus signs placed in the first character positions of the format cause the appropriate sign to be output immediately to the left of the most significant nonzero digit of the datum. The second through last plus signs may be used as digit positions similar to a # sign as required by the size of the item.

Examples:

```
PRINT USING '++##.##', 1Ø.4Ø
```

Results in the output: +1Ø.4Ø

```
PRINT USING '++++.##', 15.9Ø
```

Results in the output: +15.9Ø

One or more minus signs have an effect similar to that described above. However, a positive datum will be preceded by a blank instead of a + sign.

Examples:

```
PRINT USING '-##.##', 20.0

     Results in the output:  20.00

PRINT USING '-#,###', -705

     Results in the output:  -  705
```

A dollar sign($): One or more dollar signs in a format string cause a dollar sign to be placed at the appropriate position in the printed item. This position will depend on other format characters included in the format string. For example:

```
>PRINT USING '$###,###.##', 4600
$  4,600.00
>PRINT USING '-$##.##', 40.325
 $40.32
>PRINT USING '$##.##', 40.325
$40.32
>PRINT USING '-$$,###.##', -70
-  $070.00
>PRINT USING '+$$###.##', 70
+ $070.00
>PRINT USING '+$###.##', 70
+$ 70.00
>
```

String Fields: There are three special characters for defining string fields:

```
#   >   <
```

Table 15-3 in the Reference Section lists all string format characters and examples of using each one. The examples below illustrate the effects various combinations of these format characters have on string item format.

The pound sign (#): Each # sign in the format string represents one alphanumeric character to appear in the output. Including too few # signs causes only the specified number of characters to be printed.

```
>PRINT USING '##', 'UGANDA'
UG
```

Left angle brackets (<): in a format string cause the indicated number of characters from the left-most portion of the datum to be printed. Other format characters in the string dictate how many characters will be printed as well as the print field positions they will occupy.

```
>PRINT USING '<##', 'UGANDA'
UGA
>PRINT USING '###', 'YES'
YES
>PRINT USING '<###', 'YES'
YES
```

Right angle brackets (>): cause the specified number of characters from the right-most portion of the item to be printed. Depending on the other format string characters, the item may be printed in the first character position of the print zone, or forced to another print position as shown below.

```
>PRINT USING '>##', 'UGANDA'
NDA
>PRINT USING '>###', 'UGANDA'
ANDA
>PRINT USING '>##########', 'YES'
         YES
```

Placing more than one right or left angle bracket in the same format string has the same effect as only one bracket:

```
>A$='UGANDA'
>PRINT USING '>>##', A$
A
>PRINT USING '>>', A$
A
>PRINT USING '<', A$
U
>PRINT USING '<<#', A$
U
>PRINT USING '<>##', A$
U
```

The following program demonstrates several uses of the PRINT USING statement. User input is underlined for clarity.

```
10 REM EXAMPLE TO ILLUSTRATE PRINT USING
20 !
30 INPUT A,B,C
40 E$= 'STRING'
50 PRINT USING '<####################', E$
60 PRINT USING '>####################', E$
70 PRINT
80 F$='-##.##'
90 PRINT USING F$,A,B,C
100 PRINT USING '$$#####.##',A,B,C
110 PRINT USING '>########## EXPRESSION', E$
120 REM NOTE RESULT PLACED IN SPECIFIED FIELD
125 PRINT
130 INPUT X
135 PRINT USING '-##.##',SQR(X)
>RUNNH
!12,13,14
STRING
                STRING

 12.00
 13.00
 14.00
 $00012.00
 $00013.00
 $00014.00
     STRING EXPRESSION

!46
  6.78
STOP AT LINE 135
```

If a value for A,B or C is too large to fit in the specified field, a row of asterisks appears when the relevant PRINT USING statement tries to print the over-large value.

Example:

```
>RUNNH
!1200, 45,7
STRING
                STRING

******
 45.00
  7.00
 $01200.00
 $00045.00
 $00007.00
     STRING EXPRESSION

!1456
 38.16
STOP AT LINE 135
```

Changing Output Line Length

By using the MARGIN statement, the length of the output line can be altered. Unless a MARGIN statement is included in the program, the output line is assumed to be 80 characters. The choice of line length depends on the terminal and can be any number of characters from 1 to 32767.

A BASIC/VM program can have any number of MARGIN statements. The specifications set up by the first MARGIN statement will remain in effect until a subsequent MARGIN statement or a MARGIN OFF statement is encountered. MARGIN OFF turns off all previously set margins, leaving only the default line length of 80 characters in effect.

The following program sets the output line length to 45 characters:

```
10  REM OUTPUT A MATRIX USING MAT PRINT
20  MARGIN 45
30  DIM M(2,6)
40  MAT READ M
50  MAT PRINT M
60  DATA 1,2,3,4,5,6,7,8,9,10,11,12
70  REM EACH COMMA UP TO 9 MEANS TAB 22 CHARACTERS
80  REM THE NUMBERS AFTER 9 ARE SEPARATED BY 21 CHARACTERS
```

The following results are obtained when the program is run:

```
1                    2                    3
4                    5                    6
7                    8                    9
1Ø                   11                   12
```

This program sets the output line to 4Ø characters:

```
1Ø REM OUTPUT A MATRIX USING MAT PRINT
2Ø MARGIN 4Ø
3Ø DIM M(2,6)
4Ø MAT READ M
5Ø MAT PRINT M
6Ø DATA 1,2,3,4,5,6,7,8,9,1Ø,11,12
```

Results in:

```
1                    2
3                    4
5                    6
7                    8
9                    1Ø
11                   12
```

SECTION 6

PROGRAM CONTROL STATEMENTS

INTRODUCTION

Program control statements establish the order in which program statements are to be executed. These control statements direct program branching (e.g., GOTO, GOSUB), define loops (e.g., FOR-NEXT), transfer control to other programs (e.g., CHAIN, COMINP), and tell a program when to stop (e.g., STOP, END).

There are three categories of control statements: conditional, unconditional and loops. Conditional statements transfer program control on the basis of a specified condition evaluating to true or false. If a condition is true, one set of statements is executed; if false, an alternative path is taken. Unconditional statements affect execution control independent of conditions established by other statements. Loop statements cause a program to loop or repeat a section of code until a specified condition is attained. Below is a list of statements within each group.

Statement Type	Statements Used
Unconditional	GOTO, CHAIN, COMINP STOP, END GOSUB-RETURN
Conditional Structures	IF-THEN-ELSE, DO-DOEND ON-{GOTO / GOSUB}
Loops	FOR-{WHILE / UNTIL}-NEXT

STATEMENT MODIFIERS

The statement modifiers IF, WHILE, UNTIL and UNLESS can be used with any executable statement to establish conditions under which the statement should be executed. Unconditional statements can be made conditional with statement modifiers, increasing control structure flexibility. Below is a list of modifiers and their respective effects on companion statements. The general format of this statement - modifier combination is:

statement modifier condition [modifier condition]*

statement is an executable statement; condition is a logical expression; * means repeat as necessary, and modifier is one of the following:

- IF - execute the statements if condition is true.
- UNLESS - execute the statement if condition is false.
- UNTIL - execute the statements repeatedly while condition is false.
- WHILE - execute the statement repeatedly while condition remains true.

More than one modifier may be included in a statement line. They are processed from right to left.

BRANCHING WITHIN A PROGRAM

There are two types of unconditional statements: those which cause branching within a single program, and those which branch to external programs.

The STOP and END statements do not cause branching; instead they affect program control by terminating program execution regardless of any previously set conditions.

Transfer To Another Statement

One internal unconditional statement, GOTO transfers execution control directly to a specified statement line regardless of the value of any condition. The transfer may be either forward or backward.

Example:

```
10  INPUT A
20  GOTO 50
30  A = SQR(A+14)
50  PRINT A, A*A
60  IF A<20 GOTO 30
```

In this program segment, execution control is unconditionally transferred to line 50 at line 20. At line 60, program control is transferred to line 30, if a certain condition (i.e., A<20) is true; if the condition is not met, executions continues with the next sequential statement. If a GOTO transfers control to a previous statement, a loop can be created as in the following example. Should the GOTO, be unconditional, an infinite loop may be established.

Example:

```
100  PRINT 'INITIAL VALUE'
110  INPUT I
120  PRINT 'TYPE CHANGE'
130  INPUT C
140  REM C IS + OR -
150  IF C=0 THEN 200
160  I=I+C
170  PRINT 'NEW VALUE IS', I
180  PRINT
190  GOTO 120
200  STOP
```

The section between 120 and 190 is repeated until the value of C is equal to 0.

Transfer to Internal Subroutine

Like GOTO, the GOSUB statement transfers control directly to a statement line number. This line is generally the beginning of a multi-line subroutine which must always end with a RETURN statement. The RETURN statement transfers program control back to the statement following GOSUB which called it, and program execution continues. For example, the following program conditionally transfers control to a subroutine on the basis of a value input from the terminal:

Example:

```
5 PRINT 'INPUT A VALUE FOR A'
1Ø INPUT A
2Ø IF A<2Ø THEN GOSUB 4Ø
3Ø GOTO 8Ø
4Ø PRINT 'A LESS THAN 2Ø'
5Ø A=COS(A)
6Ø PRINT 'COSINE OF A =': A
7Ø RETURN
8Ø PRINT 'FINAL VALUE OF A =': A
9Ø END
>RUNNH
INPUT A VALUE FOR A
!23
FINAL VALUE OF A = 23
>RUNNH
INPUT A VALUE FOR A
!12
A LESS THAN 2Ø
COSINE OF A = .8438539587325
FINAL VALUE OF A = .8438539587325
```

This program illustrates the use of GOSUB-RETURN to set up two alternate execution paths. If A is <2Ø, the subroutine is executed; if A> or = to 2Ø, control transfers directly to line 8Ø, then ENDs.

BRANCHING TO EXTERNAL PROGRAMS

The COMINP and CHAIN statements direct the flow of a program to external command files or programs.

Control Transfer to Command Files

The COMINP statement, followed by a quoted argument (e.g., pathname), stops the current program flow, calls the specified external file (called command file), to the foreground, then reads and executes the commands in it. The command file essentially takes the place of input from the terminal. This is useful when repeated execution of a series of commands is required by one or more programs. The series of commands and associated data can be put in a command file and then be called by any program as needed.

The BASIC/VM COMINP statement is much like the PRIMOS command COMINPUT (see Appendix D). The argument following a COMINP statement must be a legal BASIC string. COMINP may also be used as a command, and, as such, takes an unquoted string argument.

The following program, TESTRUN, uses the COMINP statement to call an external program, TEST to the foreground. BASICV reads commands from this program until instructed by the command COMINP TTY to resume accepting commands from the terminal. This must be the last command in the external file. The commands COMINP PAUSE and COMINP CONTINUE can be used to temporarily halt and then continue the process of the command file.

Example:

The command file 'TEST' was created under the PRIMOS EDITOR and exists in the same UFD as the BASICV program 'TESTRUN'. Both are listed below:

TESTRUN

```
OK, slist TESTRUN
GO
10 PRINT 'TESTRUN'
20 COMINP 'TEST'
```

TEST

```
OK, slist TEST
GO
10 PRINT 'THIS IS A COMINP FILE, TEST'
20 INPUT A
30 PRINT A
40 IF A < 10 GOTO 20
50 PRINT 'DONE!'
RUN
1
2
3
4
5
6
7
8
91
COMINP TTY
```

The last command in the file 'TEST' is COMINP TTY which returns program control to the terminal.

Example: (continued)

The following output results when TESTRUN is executed:

```
NEW OR OLD:OLD TESTRUN
>RUNNH
TESTRUN
>1Ø PRINT 'THIS IS A COMINP FILE, TEST'
>2Ø INPUT A
>3Ø PRINT A
>4Ø IF A < 1Ø GOTO 2Ø
>5Ø PRINT 'DONE!'
>RUN
TESTRUN        TUE, SEP Ø5 1978        14:54:31

THIS IS A COMINP FILE, TEST

!1
1
!2
2
!3
3
!4
4
!5
5
!6
6
!7
7
!8
8
!91
91
DONE!
STOP AT LINE 5Ø
>COMINP TTY
>
```

At this point, the BASIC/VM prompt is returned, indicating that it is ready to accept input from the terminal.

Transferring Control to an External Program

A program external to the one currently in the foreground may be executed by including a CHAIN statement in the foreground program. When the CHAIN statement is encountered, execution of the foreground program is halted, all currently open files are closed, all variables and arrays are deallocated, and the specified external program is loaded into the foreground and then executed. This external file may be either a source or binary (compiled) file. The CHAINED program runs until an END or any other control-transfer statement (e.g., CHAIN) is encountered. CHAIN is useful in two situations:

1. If a single program is too large to be loaded into memory at one time, it can be divided into more than one program, each one being loaded in separately with CHAIN.

2. A particular program may be used by several others by including CHAIN statements in each calling progam.

Example:

> Assume that in a directory, there are three programs named PROGA, SORT, and OUTPUT respectively. PROGA opens and writes data to a file called COMMON; when the CHAIN 'SORT' command in PROGA is encountered, SORT is called to the foreground and executed. SORT then writes to a data file the values to be used by the file OUTPUT, then conditionally CHAINs to OUTPUT.

```
 1Ø  ! PROGA - MAIN ROUTINE
1ØØ  DEFINE FILE #1 = 'COMMON', ASC SEP
11Ø  INPUT A, B, C
       .
       .
16Ø  M$ = 'DATA'          ! DATA FILE FOR SORT
       .
       .
2ØØ  WRITE #1, A, L, M$   ! WRITE OUT PARAMETERS FOR SORT
       .
       .
 999 CLOSE #1
1ØØØ  CHAIN 'SORT'
```

BASIC/VM locates SORT, closes PROGA and loads SORT. It then executes SORT beginning with the lowest-numbered line, in this case, line 1ØØ.

```
100  REM THIS PROG SORTS DATA, CHAINS TO OUTPUT
110  DEFINE FILE #1= 'COMMON', ASC SEP
120  READ #1, A, L, M$    ! READ PARAMETERS FOR SORT
130  DEFINE FILE #2 =M$   ! OPEN DATA FILE
       .
       .
500  REWIND #1
510  WRITE #1, A, L, Z, X$  ! WRITE OUT PARAMETERS
515  !FOR OUTPUT ROUTINE
520  IF (A = 1) THEN CHAIN 'OUTPUT'
```

If the value of A is equal to 1, the file OUTPUT is brought to the foreground and executed.

CONDITIONAL PROGRAM BRANCHING

Conditional statements generally operate in pairs or groups: the first statement sets a condition and the second statement provides an executable alternative depending on the value of the condition. The IF statement, for example, is used in conjunction with other statements such as GOTO, THEN, ELSE, and DO to establish conditional branches for program control to follow.

Single Condition Branching: IF Structures

There are three general formats for the IF statement. These formats allow control to be directed to a single statement, to a series of statements (subroutine), or to an alternate statement or subroutine depending on whether a conditional expression is true or false.

Format 1: sets up two alternate paths for the program to take based on whether the indicated logical expression (condition) is true or false.

```
IF expr THEN {stmt   } [ELSE {stmt-2 }]
             {lin-num}       {lin-num}
```

expr is a logical expression which is evaluated to true or false; stmt is a legal BASIC statement and lin-num is a line number of a statement in the program.

Format 2: is a simplified version of the previous format, designating one line number to which control will be transferred if a condition is true. If the expression evaluates to false and no ELSE clause is included, the next sequential statement is executed. statement is executed.

```
IF expr GOTO lin-num [ELSE lin-num]
```

Parameters are the same as in the previous format.

Format 3: sets up a multi-branched conditional structure:

```
IF log-expr THEN DO
        .
        .
        .
      DOEND
    [ELSE DO  ]
    [   .     ]
    [   .     ]
    [   .     ]
    [DOEND    ]
```

In this structure, if the logical expression (log-expr) is evaluated to be true, the statements in the DO...DO END block are executed. If the expression evaluates to false, the statements in the ELSE DO...DOEND block are executed. If no ELSE DO clause exists, the next sequential statement is executed whether or not the DO...DOEND block has been executed.

Examples:

```
200  IF A$ = 'REENTER' THEN DO
210    M (I,J) = 6
220    J = J-1
230  DOEND
240  ELSE DO
250    M (I,J) = K
260    J = J+1
270    PRINT J
280  DOEND
```

IF may be used in conjuction with one or more statements, as indicated in the program. The series of IF statements first sets up a condition specifying a string value for A$. If that condition is true, the program is instructed to THEN DO the subsequent statements until a DOEND is encountered. If the value of A$ does not equal the specified string, the program is instructed to ELSE DO the subsequent statements until a DOEND is again encountered. A THEN DO or ELSE DO statement is always used in conjunction with a DOEND statement.

Branching on Multiple Conditions

The ON-GOTO and ON-GOSUB statement pairs set up one or more conditions for control transfer, by means of an arithmetic expression, and a corresponding set or list of line numbers to which control will be transferred when one of the indicated conditions occurs. Conditions are set up by arithmetic expressions which evaluate to integer values.

Format 1: transfers control to a statement indicated by a line number in the ON-GOTO statement.

```
ON expr GOTO lin-num-1 [,lin-num-2 - lin-num-n]
```

expr is an arithmetic expression which is evaluated and truncated to an integer. If the result is 1, control transfers to the first line number listed. If the result is 2, control transfers to lin-num-2, and so forth.

Example:

```
ON I GOTO 100, 200, 450
```

I is evaluated and truncated to yield an integer less than or equal to the number of statement lines listed with GOTO (i.e., 3). If I is evaluated to greater than 3, a GOSUB OVERRANGE error message is returned.

The ON-GOTO combination essentially operates like several IF statements. For example, if 500 is the last statement in the program, the previous ON-GOTO statement could be replaced by the following series of IF statements:

```
 .
 .
 .
40  IF I < 1 GOTO 500
50  IF I > 3 GOTO 500
60  IF I = 1 GOTO 100
70  IF I = 2 GOTO 200
80  IF I = 3 GOTO 450
 .
 .
 .
500 END
```

In this case, the 'GOTO 500' statements are necessary to prevent an out-of-range error if the value of I is anything but 1, 2, or 3.

Format 2: transfers control to a subroutine beginning at the line number chosen depending on the expression value:

```
ON expr GOSUB lin-num-1 [,lin-num-2,...lin-num-n]
          .
          .
          .
          .
        RETURN
```

expr, an arithmetic expression, is evaluated and truncated to an integer. Control transfer works as in the ON-GOTO statement. When a RETURN statement is encountered in the subroutine, control returns to the statement immediately following the ON-GOSUB statement. For every GOSUB executed in a program, exactly one RETURN must be executed.

LOOP STATEMENTS

Creating Simple Loops

The FOR and NEXT statements together create a loop, or a series of statements that are executed repeatedly until a previously determined condition is met. The FOR statement begins the loop by initializing a variable and setting a limit on its value. The NEXT statement ends the loop and directs the program back to FOR at which point the variable is incremented by one (unless otherwise specified). The FOR-NEXT loop continues until the value of the variable has reached the set limit. The format is:

```
FOR index = start TO end [STEP incr]
   .
   .
   .
 NEXT
```

index is a numeric variable representing the loop index. It is initialized to start, a numeric expression; the loop is extended until the end value for the index is reached. incr represents the increment value; default = 1.

Example:

```
15  PRINT 'X', 'X*2', 'X^2'
20  FOR X=1 TO 10
30  PRINT X, X*2, X^2
40  NEXT X
50  END
```

This program initializes the value of X to 1 (line 20), PRINTs the value, its double, and its square (line 30), returns control to line 20 and increments the value of X to 2. The loop continues until the value of X equals 10. When this happens, the program skips the NEXT X statement line 40 and stops at line 50.

It is also possible to specify an increment value other than 1. This is accomplished by including STEP in the FOR statement. In the program above, the values of X can be set from 1 to 100 with increments of 5 as follows:

```
10 PRINT 'X', 'X*2', 'X^2'
20 FOR X=1 TO 100 STEP 5
30 PRINT X, X*2, X^2
40 NEXT X
```

In this case, the value of X is initialized to 1, incremented to 6, and is incremented by 5 with each pass through the loop until the value is equal to or greater than 100.

Note

The STEP value may be any value. Also, in FOR-loops with statement modifiers, the step size is assumed to be zero unless otherwise specified.

Conditional Loops

The statement modifiers WHILE and UNTIL may be used with the FOR- NEXT statements to place special conditions on loop execution. Instead of assigning an end value to index, (the variable which is incremented during loop execution), the loop is executed WHILE, or UNTIL a specified condition exists.

WHILE: causes loop execution and variable incrementation to continue as long as the specified condition is true.

Example:

```
X = 10
FOR I = 1 STEP 1 WHILE I<X
X = X/2
PRINT I, X
NEXT I
STOP
```

On each pass through the loop, the value of X is divided by 2. The value of I is incremented by 1 as long as it is less than the value of X. If no STEP is specified, I would be incremented by zero, or unchanged, creating an infinite loop.

UNTIL: causes loop execution and variable incrementation to continue until the specified condition is met.

Example:

```
10  FOR I = 1 STEP 1 UNTIL J = 1E4
20  J = J*2 + TAN(I)
30  NEXT I
40  END
```

On each pass through the loop, the value of J is squared and added to the function of I. The value of I continues to be incremented by 1 until the value of J is equal to 10,000.

SECTION 7

EDITING AND DEBUGGING

INTRODUCTION

Errors in a program are basically of three types: syntax errors, which are violations of language rules: execution errors, which occur when a program attempt is illogical or impossible action (sometimes fatal): and logic errors, or faults in program logic which produce strange results, possibly including program failure. Errors can be detected by inspection, compilation and execution, as explained in Section 3. Obvious errors in a program can be easily corrected with simple editing procedures (discussed in Section 3), or with the editing commands detailed in this section. Inconspicuous logic errors which cause a program to execute improperly or not at all may be more difficult to detect. The BASICV debugging tools presented here may be of use in locating such errors.

EDITING A BASIC/VM PROGRAM

In addition to the simple edit features discussed in Section 3, BASIC/VM provides commands to perform the following edit functions:

- deleting one or more statements from a program (DELETE, EXTRACT)
- editing individual lines (ALTER)
- renumbering statements after edit (RESEQUENCE)
- determining number of lines in a file (LENGTH)

Deleting Specific Lines

The DELETE command can be used to remove specific statement lines from a program. The format is:

DELETE lin-num-1[,...{lin-num-n / lin-num-i - lin-num-n}]

Example:

```
DELETE 100, 130-160, 195
```

This deletes line 100, lines 130 through 160 (inclusive), and line 195.

Extracting Statement Lines

The EXTRACT command allows the user to delete all lines in a program except those specified. The format is:

EXTRACT lin-num-1[,...{lin-num-n / lin-num-i - lin-num-n}]

For example, to delete all lines in a program except 10-50 (inclusive), and line 59, type:

```
EXTRACT 10-50, 59
```

This will delete all lines in the program except those indicated.

Editing Within Lines

Instead of deleting and retyping a line completely, it is possible to modify a portion of it. The ALTER command provides a series of subcommands which enable editing within lines. The ALTER subcommand mode is entered by typing:

```
ALTER line-number
```

where line-number is the line to be modified. A complete list of ALTER subcommands can be found in Section 15. Here are some of the more useful ones:

- Cx — Copies line up to but not including x, where x is any character.
- Dx — Deletes line up to but not including x.
- F — Copy to end of line.
- I/str/ — Insert string (str) at current position.
- R/str/ — Retype line with string from current position.
- Q — Exit from alter mode.

Using ALTER

The following example shows how the subcommands are used. They are entered in response to the ':' prompt and several may be packed on a line without delimiters. ALTER returns the colon after every (CR), allowing as many chances as you need to modify the line. Type Q to return to BASIC/VM command level.

Examples:

1.
```
> ALTER 100
100 IF X=Y GOTO 230
: C=E1I/>/F                    >Copy up to, but not
                                including the = sign.
                                Erase 1 character.
                                Insert >.
                                Copy the rest of the line.
100 IF X>Y GOTO 230

:Q                             Another chance to ALTER.
                               Leave ALTER 'mode'.
```

2.
```
>ALTER 230
230 PRINT 'TOO LOW'
: M11E3A/HIGH'/                >Move 11 characters..
                                Erase 3 characters.
                                Append HIGH' to the end
                                of the line.
230 PRINT 'TOO HIGH'

:Q                             Another chance to ALTER.
                               Leave ALTER 'mode'.
```

Fixing a Simple Program

This example shows the process of editing, compiling, and executing a new program:

```
10 ! THIS PROGRAM DEMONSTRATES THE  USE OF AN ACCUMULATOR
20 ! D= ACCUMULATED DEPOSITS
30 !X= DEPOSITS; N= NUMBER OF DEPOSITS
40 D=0
45 N=0
50 READ X
60 D= D+X ! USE OF LET IS OPTIONAL
70  N=N +1  ! THE ACCUMULATOR
75 PRINT 'TOTAL AMOUNT OF DEPOSITS TO DATE IS; $ '; D
80 PINT 'NUMBER OF DEPOSITS', N
90 GOTO 50
100 DATA 14.15, 234.56, 78.90, 12.00, 0
>COMPILE
80 PINT 'NUMBER OF DEPOSITS", N
INVALID WORD IN STATEMENT
>ALTER 80
80 PINT 'NUMBER OF DEPOSITS', N
:CII/R/F
80 PRINT 'NUMBER OF DEPOSITS', N
:Q
>COMPILE
>EXECUTE
TOTAL AMOUNT OF DEPOSITS TO DATE IS; $ 14.15
NUMBER OF DEPOSITS   1
TOTAL AMOUNT OF DEPOSITS TO DATE IS; $ 248.71
NUMBER OF DEPOSITS   2
TOTAL AMOUNT OF DEPOSITS OF DATE IS; $ 327.61
NUMBER OF DEPOSITS   3
TOTAL AMOUNT OF DEPOSITS TO DATE IS; $ 339.61
NUMBER OF DEPOSITS   4
TOTAL AMOUNT OF DEPOSITS TO DATE IS; $ 339.61
NUMBER OF DEPOSITS   5
END OF DATA  AT LINE  50

>65 IF X=0 GOTO 110       (110 is END statement)
110 END                  (END statement removes 'END OF DATA' message)
>COMPILE
>EXECUTE
TOTAL AMOUNT OF DEPOSITS TO DATE IS; $ 14.15
NUMBER OF DEPOSITS   1
TOTAL AMOUNT OF DEPOSITS TO DATE IS; $ 248.71
NUMBER OF DEPOSITS   2
TOTAL AMOUNT OF DEPOSITS TO DATE IS; $ 327.61
NUMBER OF DEPOSITS   3
TOTAL AMOUNT OF DEPOSITS TO DATE IS; $ 339.61
NUMBER OF DEPOSITS   4
>85 PRINT                 (makes printout format neater)
>COMPILE
```

```
>EXECUTE
TOTAL AMOUNT OF DEPOSITS TO DATE IS; $ 14.15
NUMBER OF DEPOSITS   1

TOTAL AMOUNT OF DEPOSITS TO DATE IS; $ 248.71
NUMBER OF DEPOSITS   2

TOTAL AMOUNT OF DEPOSITS TO DATE IS; $ 327.61
NUMBER OF DEPOSITS   3

TOTAL AMOUNT OF DEPOSITS TO DATE IS; $ 339.61
NUMBER OF DEPOSITS   4
```

Determining the Total Number of Statements

The LENGTH command can be used to determine the number of lines in the foreground file. Its format is:

```
LENGTH
```

Example:

```
>LENGTH
25 LINES
```

Renumbering a Program:

After deleting and inserting statement lines in a program, it may be necessary to renumber them in a logical sequence. The RESEQUENCE command renumbers a program with default values or with supplied values. Any BASICVM program can be renumbered with RESEQUENCE. The format is:

```
RESEQUENCE [new-start, old-start, new-incr]
```

where new-start is the number with which to begin renumbering; old-start is the line at which to begin the resequence and new-incr is the new increment value with which to continue renumbering. If no values are specified, the default values are 1ØØ, 1, 1Ø.

Example:

```
> DELETE 20, 40, 80-120
> LISTNH
10 PRINT
30 PRINT 'X', 'X^X', 'X*X'
50 LET X=9
60 PRINT
70 PRINT X, X^X, X*X
130 END

> RESEQUENCE 10, 10, 5
> LISTNH
10 PRINT
15 PRINT 'X', 'X^X', 'X*X'
20 LET X=9
25 PRINT
30 PRINT X, X^X, X*X
```

Renumbering begins with number 10, at current program statement 10, in increments of 5.

DEBUGGING A PROGRAM

Control errors in a program are sometimes difficult to locate. Finding and rectifying these errors in a program is called debugging. The debugging process can be simplified through the use of the commands BREAK and TRACE, and the ON ERROR and PAUSE statements.

Debug Commands

The BREAK ON command sets up halts, or breakpoints, at specific lines in a program. These breaks return the user to BASICV command level. Values being passed within a program can be displayed at these breakpoints. Program execution is resumed only if CONTINUE is typed. BREAK ON is issued immediately following compilation and prior to execution. The format is:

BREAK {ON / OFF} lin-num-1 [,...lin-num-n]

BREAK OFF, typed prior to re-execution, can turn off any or all previously set breakpoints. If no line numbers are specified with BREAK OFF, all breakpoints are eliminated.

The following source program demonstrates how to use break points to cheat at a computer guessing game.

```
10 PRINT 'WHAT NUMBER AM I THINKING OF'
15 N=INT(50*RND(0)+1)
20 FOR C = 0 TO 10
30 INPUT X
50 IF X<N GOTO 90
60 IF X>N GOTO 110
70 PRINT 'RIGHT, ANOTHER!'
80 GOTO 140
90 PRINT 'TOO LOW'
100 GOTO 120
110 PRINT 'TOO HIGH'
120 NEXT C
130 PRINT 'TIME IS UP.  ANOTHER'
140 INPUT A$
150 IF LEFT(A$,1)='Y' THEN 15
160 STOP


>COMPILE
>BREAK ON 50
>EXECUTE


WHAT NUMBER AM I THINKING OF
!  27
BREAK AT LINE 50
>PRINT N
41
>X=41
>CONTINUE
RIGHT, ANOTHER!
!N
STOP AT LINE 160
>BREAK OFF 50
```

The program is instructed to stop at line 50. The user finds out what the number is (PRINT N) and sets an answer (X) equal to that number, in this case, 41. Program execution is resumed with the CONTINUE command at which point the system responds that the correct answer has been given. When the answer 'N' (for 'NO') is given to the 'RIGHT, ANOTHER!' prompt, the user is returned to command level. 'BREAK OFF 50' indicates that when the program is run again, no break will occur at line 50.

Inserting Program Halts

The PAUSE statement acts as an executable BREAK. It is used in conjunction with CONTINUE. When the program halts at the line number on which the PAUSE statement appears, the words "PAUSE AT LINE x;, where x is the appropriate line number, are displayed, indicating the temporary halt. CONTINUE is typed whenever the user wishes to resume the program after a breakpoint.

Example:

```
1Ø PRINT 1
2Ø PAUSE
3Ø PRINT 3
4Ø END
>RUNNH
1
PAUSE AT LINE 2Ø
>CONTINUE
3
```

Tracing Statement Execution

The TRACE ON command is issued immediately after COMPILing, and immediately prior to EXECUTing a program. It is useful in tracing the path of program execution, thereby expiating the debugging process. The line number of each statement executed is displayed in brackets, e.g., [12Ø]. For example, if a specified condition is met, a GOTO or GOSUB statement will be executed, and its line number will be displayed. When program execution terminates, type TRACE OFF, and the program can be re-executed normally.

Example:

```
5 PRINT 'INPUT A VALUE FOR A'
10 INPUT A
20 IF A<20 THEN GOSUB 40
30 GOTO 80
40 PRINT 'A LESS THAN 20'
50 A=COS(A)
60 PRINT 'COSINE OF A =': A
70 RETURN
80 PRINT 'FINAL VALUE OF A =': A
90 END
>COMPILE
>TRACE ON
>EXECUTE
[5]
INPUT A VALUE FOR A
[10]
!13
[20]
[40]
A LESS THAN 20
[50]
[60]
COSINE OF A = .9074467814502
[70]
[30]
[80]
FINAL VALUE OF A = .9074467814502
[90]
>TRACE OFF
>EXECUTE
INPUT A VALUE FOR A
!23
FINAL VALUE OF A = 23
```

Notice that after TRACE OFF was typed, the program executed without line number display.

TRAPPING EXECUTION ERRORS

There are several ways to set up error traps within a program. The ON ERROR statement establishes a line number to which control will be transferred when a run-time error occurs. Variation of the ON ERROR statement provides for redirection of program flow if an I/O error occurs on a specified unit. For example, if invalid data is input, control can be transferred to a statement that will print out an appropriate error message. The format is:

```
ON ERROR [#unit] GOTO lin-num
```

The #unit option is used in trapping I/O errors on a unit previously opened by a DEFINE FILE statement. See Section 8 for details.

Turning Off Error Traps

The ERROR OFF statement can be employed to cancel all error traps established by ON ERROR GOTO statements. The format is:

```
ERROR OFF
```

The example below uses the ERROR OFF functionality to turn off error trapping after a certain point in the program.

Identifying Locations and Codes of Errors

The following special variables and functions can be used to identify the location and nature of errors trapped:

ERR	a variable set to the code number of the trapped error.
ERL	a variable set to the line number at which an error occurred.
ERR$(num-expr)	a function which outputs the text of the error message associated with an error code, represented by num-expr.

A complete list of run-time error codes and corresponding messages can be found in Appendix C.

Using Error Traps

The following example uses several of the error trap features presented above:

```
110 INPUT 'FILENAME ', A$
100 ON ERROR GOTO 1000
120 DEFINE FILE #1 = A$
130 DEFINE FILE #2 = 'OUTFILE'
      .
      .
      .
500 ERROR OFF
      .
      .
      .
990 STOP
1000 PRINT ERR$(ERR); 'AT LINE' : ERL
1010 CLOSE #1,#2
1020 STOP
```

This program directs control to line 1000 if an error occurs between line 110 and line 500. Line 1000 prints the text of the error via the ERR$(ERR) function. It also identifies the error line via the ERL function. Line 1010 instructs the program to close the opened files. The program then stops at line 120. If no error occurs, the program will stop at line 990. If the ERROR OFF statement is omitted, the ON ERROR will be effective throughout the entire program.

The statement to which control is transferred may also identify the error by its code, and then instruct the program what to do next.

For example:

```
IF ERR = 13 THEN DO
   PRINT LIN(2), 'END OF TEST'
   END
DOEND
```

indicates that if an END OF DATA error, (error code 13), has occurred, the message 'END OF TEST' will be printed, and the program will stop.

SECTION 8

FILE HANDLING

INTRODUCTION

BASIC/VM utilizes all of the file types provided by the File Management System (FMS) of PRIMOS. These file types, and the type-codes by which they are identified, are:

- ASCII sequential (ASC, the default)
- ASCII sequential separated (ASCSEP)
- ASCII sequential line-numbered (ASCLN)
- ASCII direct access (ASCDA)
- Binary sequential (BIN)
- Binary direct access (BINDA)
- Segment directory (SEGDIR)
- Multiple Index Data Access (MIDAS)

Because these files are all created under the auspices of FMS, file compatibility between BASICV programs and programs written in other Prime languages (e.g., COBOL, interpretive BASIC, FORTRAN) is assured.

Implementation of these data files in BASIC/VM programming is known as 'file handling'. File handling usually involves opening (or defining) a file, writing data to the file for storage, and reading, or retrieving, the data when needed.

The BASIC/VM default file type, ASCII sequential, provides storage and access methods suitable for most programming needs. The seven other file types offer alternate features which may optimize storage efficiency and/or program execution when properly utilized.

This section describes the available file types, their features, possible uses, and the statements needed to perform the routine file handling operations, listed below. Only the basic information needed to use these files in routine BASIC/VM programming is contained in this section. Users requiring more details on file properties, features and uses should consult Appendix E, Advanced File Handling.

Operation	Statement Used
• Opening a file	DEFINE
• Writing data to a file	WRITE
• Examining data storage	TYPE, SLIST
• Reading data from a file	READ[*], READLINE
• Moving the file pointer	POSITION, REWIND
• Updating record data	WRITE
• Trapping errors during file operations	ON {ERROR | END} GOTO
• Write a matrix to a file	MAT WRITE
• Reading data into a matrix	MAT READ[*]
• Closing a data file	CLOSE
• Deleting a data file	REPLACE

OPENING A DATA FILE

The DEFINE FILE statement is the key to all data file operations in BASIC/VM. DEFINEing a file is relatively simple but involves a number of concepts which are important to all file handling operations. The DEFINE process does several important things including:

- reserving buffer space in memory for data storage
- naming a file
- assigning a particular file type to the file
- optionally changing the record size of the file
- restricting I/O operations to reading or writing

The DEFINE Statement

The format of the DEFINE statement is:

```
DEFINE  READ    FILE #unit = filename  [,type-code] [,record-size]
        APPEND
```

The parameters are discussed below.

File units

PRIMOS requires some buffer space in physical memory to serve as an intermediary storage area for each opened file. These buffers are called file units. To open or define a file in BASIC/VM, a correlation must be established between a filename and a file unit number. The file unit number is specified by the unit parameter, a numeric expression with a range of 1 to 12. The number assigned to the file is used as a sort of shorthand reference to the file throughout subsequent file operations. The # sign is a required part of the statement proper, and signifies that a data file is to be opened on the specified unit.

Up to 12 file units may be opened and active at one time per user in the BASICV subsystem. If an attempt to open more than 12 units is made, an error message will be displayed.

Filename

Each data file opened must be assigned a name, as represented by filename, a legal BASIC string parameter.

Type-code

As mentioned in the introduction, each file type available under FMS has a particular type-code by which it is identified to BASIC/VM and PRIMOS. All file types, their corresponding type-codes, and important features, are listed in Table 8-1. Note that specification of type-code is optional. The default type is ASCII sequential (ASC).

Record-size

Items in a data file are stored in logical units called records. The size of a record determines how many characters it can contain. This character limitation is measured in words at the rate of 2 characters per word. The default record-size is 6Ø words, or 12Ø characters. The record size of a file may be increased or decreased by specifying the appropriate numeric value for the record-size parameter. record-size is specified in number of words per record, as opposed to number of characters. The minimum record-size is four words for every file type except MIDAS. The maximum record size is 512 words.

In some types of files, all records in the file are fixed to the specified number of words, or to the default size, if the record-size parameter is omitted. Records in this type of file are called fixed-length records. Each record in the file is the same length, even though each record may not contain the same number of data characters. Other types of files have variable-length records, in which each record is only as long as the data it contains.

Access Restrictions

The optional arguments, READ and APPEND, place restrictions on I/O operations that can be peformed on a file. The READ argument allows the file to be read from only. No data can be written to the file while the restriction is in effect. The APPEND argument postions the read pointer to the bottom of the file when it is opened. Each file has a pointer which keeps track of the record currently positioned to for reading or writing. This restriction allows data to be written to the bottom of the file only, unless the pointer is repositioned.

Table 8-1. File Type-Codes

Type-Code	Access Method	Contents
ASC (default)	SAM	ASCII data, formatted like terminal output, using BASICV PRINT conventions, e.g., commas, colons and semi-colons, all dictate the appropriate number of spaces to be used as data delimiters. Records variable-length and easily inspected.
ASCSEP	SAM	ASCII data stored with commas inserted as data delimiters. Data are stored and read back exactly as entered. Records fixed-length, accessed sequentially.
ASCLN	SAM	ASCII data with comma delimiters, and line numbers inserted in increments of 1Ø at the start of each record. Designed to be edited at BASICV command level.
ASCDA	DAM	Similar to ASCSEP. Records fixed-length and blank-padded as necessary. Direct access method used for quick, random access to any record in the file.
BIN	SAM	Data storage transparent to user. Records are fixed-length, accessed sequentially. String data stored in ASCII code: numeric data stored in four-word floating-point form. Provide maximum precision and compactness of numeric data, but cannot be inspected by TYPE etc.
BINDA	DAM	Same as BIN but direct access method is used for random record access. Records not data-filled are zeroed out.
SEGDIR	SPECIAL	Identifies file as a segment directory. Subordinate files, identified by number, may be SAM, DAM or other SEGDIR files. An additional DEFINE is required to access a subordinate file.
MIDAS	SPECIAL	Multiple Index Data Access files. Created by Prime-supplied MIDAS utilities.

ACCESS METHODS

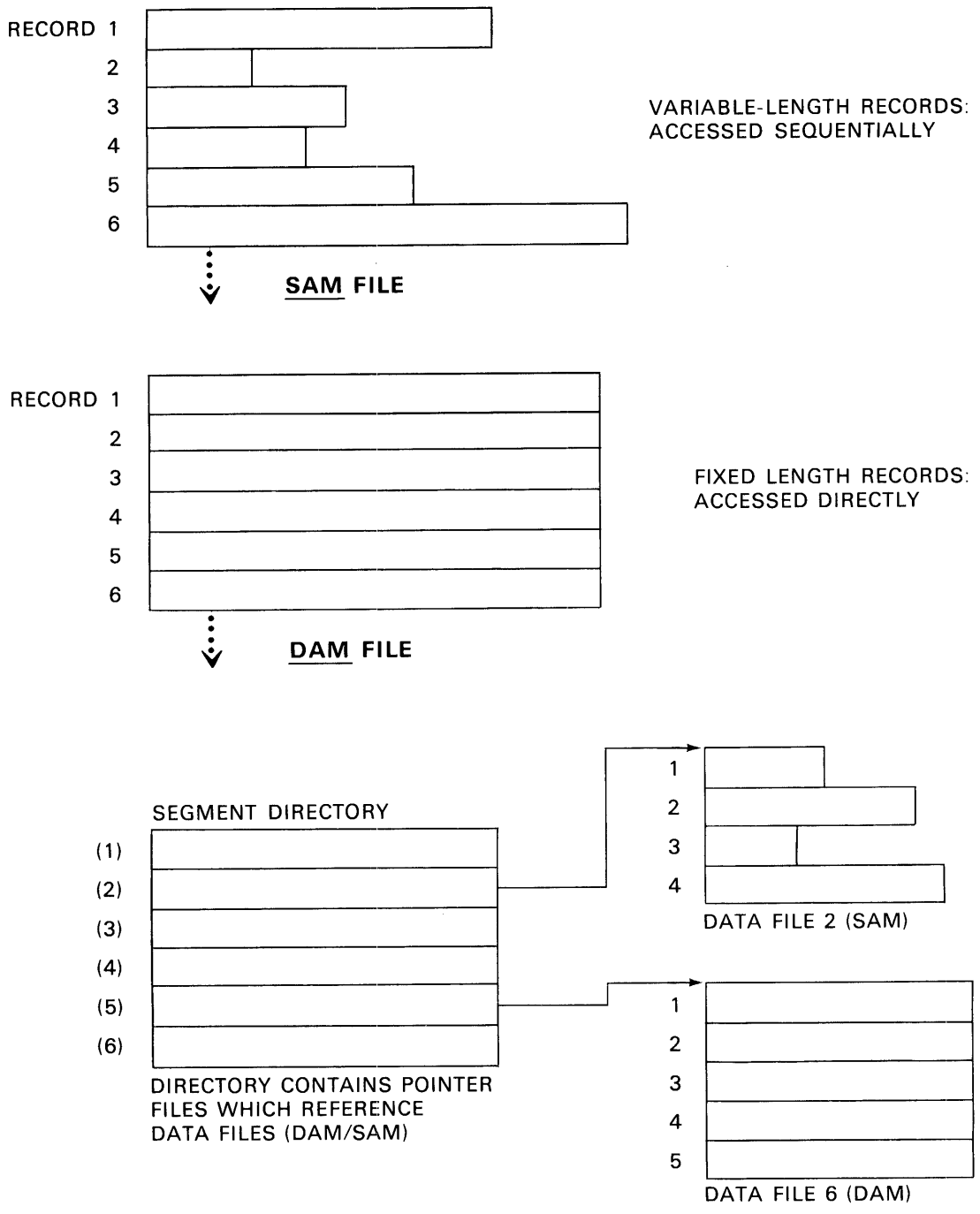
Retrieval of data from files is accomplished by one of the four access methods provided by BASICV and PMS:

- Sequential Access Method (SAM)
- Direct Access Method (DAM)
- Segment Directory Access Method (SEGDIR)
- Multiple Index Data Access Method (MIDAS)

Each access method corresponds directly to a particular file structure. These structures are illustrated in Figure 8-1. For a representation of MIDAS file structure, refer to Reference Guide, MIDAS or the Subroutine Reference Guide.

Both file structures and access methods are built into the PRIMOS operating system. Access methods determine how individual file records are identified and retrieved from their storage place on disk. The two fundamental access methods, sequential (SAM) and direct (DAM), are explained below. SEGDIR and MIDAS access methods expand upon SAM and DAM features and are discussed in the latter part of this section.

The remainder of this section is divided into four parts. Each part describes the statements used in dealing with files of a particular access type. More details on the properties of each file type, as well as extended examples of their use, can be found in Appendix E, Advanced File Handling. In particular, the properties of the default type, ASC, are discussed at length and then compared with the corresponding features of other ASCII file types. Users considering serious file handling should investigate the differences among file types before working with data files.



SEGDIR ORGANIZATION

Figure 8-1. File Structures

SAM FILE HANDLING

Sequential files can be opened and manipulated by the following set of statements.

Statements Used in Sequential Access

DEFINE	Opens, names and assigns a file type, either ASC, ASCSEP, ASCLN or BIN and associates it with a file unit.
WRITE, WRITE USING	Writes data records of the appropriate type to the opened file and advances the pointer to the next record after each WRITE.
READ [*] READ LINE	Reads the record at the current pointer position and advances the pointer to the next record. Must rewind in order to READ after a WRITE.
REWIND	Returns the pointer to the first record of the file.
ON END	Determines the action to be taken if the pointer reaches the end of the file.
CLOSE	Makes sure the file is properly restored to disk and frees the file unit for other use.

Opening a File

The first step in any file handling operation is to open or DEFINE a file. Any of the file types listed in Table 8-1 can be opened with DEFINE.

The type-codes which define SAM files are: ASC, ASCSEP, ASCLN and BIN. For example, the following statement opens an ASCII sequential file with comma separators (ASCSEP file):

```
DEFINE FILE #1 = 'ASCSEP', ASCSEP
```

Adding Data To SAM Files

Data values are written to a DEFINEd file one record at a time with successive WRITE statements. Each successive WRITE operation moves the pointer to the next sequential record, where it awaits the next instruction. Each new WRITE statement adds the indicated data to a new record in the file.

Below is an example of writing data to each sequential file type. By TYPEing each file (using the TYPE command), the data storage patterns of each file type (except binary) can be inspected.

For more details on data storage in each file type, refer to Appendix E.

Example:

```
>DEFINE FILE #1 = 'A'
>DEFINE FILE #2 = 'B', ASCSEP
>DEFINE FILE #3= 'C', ASCII
>DEFINE FILE #4 = 'D', BIN
>A=12
>B$='HELLO!'
>WRITE #1,A
>WRITE #2,A
>WRITE #3,A
>WRITE #4,A
>WRITE #1,B$
>WRITE #2,B$
>WRITE #3,B$
>WRITE #4,B$
>CLOSE #1,2,3,4
>TYPE A
12
HELLO!
>TYPE B
12,
HELLO!,
>TYPE C
  10 12,
  20 HELLO!,
>TYPE D
  `HELLO!>
```

WRITEing Formatted Data to a File

The WRITE USING statement is similar to PRINT USING (see Section 6) in that it enables the formatting of data according to a format string. Format strings are composed of special characters which are listed in Table 15-2 and Table 15-3. A summary of PRINT USING features can also be found in Section 15. Formatted data can be written to any type of ASCII file. An attempt to write formatted data to a binary file will generate an error message.

The WRITE USING statement has two formats:

```
WRITE # unit USING format-string, item-1 [,...item-n]
WRITE USING format-string, # unit, item-1 [,...item-n]
```

In either case, the format-string may be numeric or string, depending on the data, (item-1 - item-n), to be formatted.

Example:

```
>DEFINE FILE #1 = 'EXAMPLE'
>WRITE #1 USING '$###.##', 120
>WRITE #1 USING '<#####', 'FUNNY'
>WRITE #1 USING '>#####', 'FUNNY'
>WRITE #1, 120
>WRITE #1, 'FUNNY'
>CLOSE #1
>TYPE EXAMPLE
$120.00
FUNNY
 FUNNY
120
FUNNY
```

In the first WRITE USING statement, a numeric value is formatted with a decimal point, two trailing zeroes, and a dollar sign prior to the left-most digit. The pound signs (#) indicate how many digits are to be output. If the value was too large for the format string to accomodate, a string of asterisks would appear in the output. For example:

```
WRITE #1 USING '$###.##', 12000
REWIND #1
READ #1, A$
PRINT A$
*******
```

The second WRITE USING statement left-justifies a string datum in the file with by using the '<' symbol. The third statement writes the same item with right-justification by using the '>' symbol. The data written to a file with WRITE USING are stored in the format specified by the format string as shown.

READing SAM Files

Data are retrieved from SAM files with the READ statement, as shown in the previous examples. The READ statement has two variations, READ* and READLINE. All three are used to obtain information stored in a data file. Specific examples of READing each file type are included in Appendix E.

REWINDing The File Pointer

In order to READ a record above the record to which the pointer is currently positioned, a REWIND statement can be used to reposition the pointer to the top of the file. Records can not be positioned to at random in sequential files.

In sequential files, READs cannot take place immediately after a WRITE to the same file. An attempt to do so generates the following error message:

```
READ AFTER WRITE ON SEQUENTIAL FILE
```

In order to READ after a WRITE, the file pointer must be returned to the top of the file with REWIND. An alternative is to CLOSE the file, re-open it and then READ sequentially until the desired record is reached. When a file is opened, the pointer automatically positioins to the top of the file unless otherwise instructed, as with the 'APPEND' option of the DEFINE command.

Example:

```
>DEFINE FILE #1 = 'ASC'
>WRITE #1, 12
>READ #1, A
READ AFTER WRITE ON SEQUENTIAL FILE  AT LINE  Ø

>REWIND #1
>READ #1, A
>PRINT A
12
>CLOSE #1
>DEFINE FILE #1 = 'ASC'
>READ #1,A
>PRINT A
12
>CLOSE #1
```

The READ* Statement

A variation of the READ statement, READ*, forces the file pointer to remain at the current record after a READ is completed, rather than moving it to the next sequential record. This is advantageous when performing a series of READs on a single record. If a record contains several values which are to be retrieved individually during successive READs, the pointer can be 'put on hold' at the current record, enabling another READ of this record to be performed. The details of READ* are illustrated in the example below. Some points to keep in mind are:

- If a READ with no * option follows a READ*, the pointer automatically advances to the next record.
- If a READ* follows a READ*, the current record is read until all the given variables are satisfied. If necessary, the pointer then advances to the next record to satisfy any remaining variables.

Example:

```
10 DEFINE FILE #1= 'ASC*'
20 DEFINE FILE #2 = 'SEP*', ASCSEP
30 DEFINE FILE #3 = 'LN*', ASCLN
40 DEFINE FILE #4 = 'BIN*', BIN
50 REWIND #1,2,3,4
60 READ A,B,C,D,E,F
70 DATA 10,20,30,40,50,60,70
80 PRINT 'FIRST READ WITHOUT *'
90 PRINT
100 FOR N=1 TO 4
110 WRITE #N,A,B,C
120 WRITE #N,D,E,F
130 NEXT N
140 REWIND #1,2,3,4
150 FOR N= 1 TO 4
160 PRINT 'THIS IS FILE ON UNIT #': N
170 PRINT 'BEGIN READ WITHOUT *'
180 READ #N,A
190 PRINT A
200 PRINT
210 READ #N,B
220 PRINT B
230 PRINT
240 REWIND #N
250 PRINT 'NOW READ WITH *'
260 READ * #N,A
270 PRINT A
280 PRINT
290 READ * #N,B
300 PRINT B
310 PRINT
320 REWIND #N
330 PRINT 'END OF READ ON UNIT #':N
340 PRINT
350 NEXT N
360 CLOSE #1,2,3,4
370 PRINT 'END OF TEST'
380 END
>RUNNH
FIRST READ WITHOUT *
```

```
THIS IS FILE ON UNIT # 1
BEGIN READ WITHOUT *
102030

405060

NOW READ WITH *
102030

405060

END OF READ ON UNIT # 1

THIS IS FILE ON UNIT # 2
BEGIN READ WITHOUT *
10

40

NOW READ WITH *
10

20

END OF READ ON UNIT # 2

THIS IS FILE ON UNIT # 3
BEGIN READ WITHOUT *
10

40

NOW READ WITH *
10

20

END OF READ ON UNIT # 3

THIS IS FILE ON UNIT # 4
BEGIN READ WITHOUT *
10

40

NOW READ WITH *
10

20

END OF READ ON UNIT # 4

END OF TEST
```

```
>TYPE ASC*
10                  20                  30
40                  50                  60
>TYPE SEP*
10,20,30,
40,50,60,
>TYPE LN*
   10 10,20,30,
   20 40,50,60,
```

The READLINE Statement

READLINE is actually a variation of the READ statement that allows the reading of an entire ASCII file record, including commas, colons, semi-colons and spaces, as one data item. This is especially useful when default ASCII or ASCSEP files contain data with internal commas. READLINE ignores commas as data delimiters (unlike optionless READs). Thus comma-containing strings will not be broken up by READLINE. For example, if a record in an ASCII file opened on unit #1 contained the following values:

MARCUS WELBY, M.D.

READ #1, A$ would return:

MARCUS WELBY

READLINE #1, A$ would return:

MARCUS WELBY, M.D.

READ vs. READLINE

The following example emphasizes the differences between READ and READLINE for all SAM file types. Note however, that READLINE will not work on binary files.

```
10 DEFINE FILE #1 = 'ASCRL'
20 DEFINE FILE #2 = 'SEPRL', ASCSEP
30 DEFINE FILE #3 = 'LNRL', ASCLN
40 DEFINE FILE #4 = 'BINRL', BIN
50 READ A$,B,C$
60 DATA 'WELBY, MARCUS, M.D.; LIC.NO.:', 123001, 'PHONEY'
70 FOR N=1 TO 4
80 WRITE #N, A$,B,C$
90 NEXT N
100 REWIND #1,2,3,4
110 FOR I = 1 TO 4
120 PRINT 'READ FOR FILE ON UNIT #': I
130 READ #I, A$
```

```
140 PRINT
150 PRINT A$
160 REWIND # I
170 PRINT
180 PRINT 'READLINE FOR FILE ON UNIT #': I
190 PRINT
200 READLINE #I, A$
210 PRINT A$
220 PRINT
230 REWIND # I
240 NEXT I
250 CLOSE #1,2,3,4
>RUNNH
READ FOR FILE ON UNIT # 1

WELBY

READLINE FOR FILE ON UNIT # 1

WELBY, MARCUS, M.D.; LIC.NO.:              123001                PHONEY

READ FOR FILE ON UNIT # 2

WELBY

READLINE FOR FILE ON UNIT # 2

WELBY, MARCUS, M.D.; LIC.NO.:,123001,PHONEY,

READ FOR FILE ON UNIT # 3

WELBY

READLINE FOR FILE ON UNIT # 3

 WELBY, MARCUS, M.D.; LIC.NO.:,123001,PHONEY,

READ FOR FILE ON UNIT # 4

WELBY, MARCUS, M.D.; LIC.NO.:

READLINE FOR FILE ON UNIT # 4

ILLEGAL OPERATION ON BINARY FILE AT LINE  200
```

The error message displayed indicates that READLINE is not a legal operation on a binary file.

Reaching END OF FILE

When the file pointer reaches the end of a file during a READ, an END OF FILE error message is generated. This causes program execution to terminate abruptly. To avoid this, include an ON END statement to transfer control to another program line where an appropriate action will be taken. For example, the following ON END statement transfers program control to a statement which closes the file unit:

```
10 ON END #1 GOTO 200
 .
 .
 .
200 CLOSE #1
```

Trapping Errors

Errors that occur during data file access can be trapped with the ON ERROR- GOTO statement, discussed in Section 7, or with the ON END #unit - GOTO statement, designed specifically for data files. If the end of the file is reached during a file READ, for example, an error message will be generated, and the program or current operation will be halted. Inclusion of the ON END statement in a program will eliminate unwanted program halts by transferring control to another program line or by performing an appropriate action.

This statement is generally placed near the beginning of the program. It can be used for both SAM and DAM files, and is especially helpful when doing multiple READs from a binary file whose contents are not easily monitored.

Example:

```
10 DEFINE FILE #1 = 'END1', ASCSEP
20 ON END #1 GOTO 90
30 WRITE #1, 'LAND VALUES AS OF 1976'
35 WRITE #1, 'CALIFORNIA','NEW YORK'
40 REWIND #1
45 READ #1, A$
50 READ #1,B$
55 READ #1,C$
60 PRINT A$
65 PRINT
70 PRINT B$,C$
75 GOTO 120
90 PRINT 'END OF FILE REACHED'
100 REWIND #1
105 READ #1,A$,B$,C$
110 GOTO 60
120 CLOSE #1
>RUNNH
END OF FILE REACHED
LAND VALUES AS OF 1976
```

```
CALIFORNIA            NEW YORK
STOP AT LINE 120
>!IF WE REMOVE THE ON END STATEMENT, WE'RE IN TROUBLE
>20
>RUNNH
END OF FILE  AT LINE  55
```

CLOSEing a File

When access to a file is completed, it is a good idea to CLOSE the file. This ensures its proper restoration to disk, and releases the file unit on which it was opened for other use. If the file is not CLOSEd, it may be lost or truncated when a BREAK or QUIT occurs. A single CLOSE statement can close one or many file units which have been opened. Example:

```
CLOSE #1,2,3
```

Opening a Temporary File

A temporary or SCRATCH file may be opened with a short form of the DEFINE statement:

```
DEFINE SCRATCH FILE #unit [,file-type] [,record-size]
```

The indicated unit is opened as a temporary file of any type except MIDAS. When the unit is CLOSEd, the file is deleted. No filename need be specifed. The record-size can be optionally modified.

DAM FILE HANDLING

The statements used in dealing with DAM files are identical to those previously discussed under SAM file handling. The important exception is POSITION. The following is a list of all available DAM file handling statements.

Statements used in Direct Access

Statement	Description
DEFINE	Opens, names and identifies a direct access file as ASCII (ASCDA) or binary (BINDA); associates it with a file unit and optionally sets the record size (in words).
WRITE,WRITE USING	Writes data records to the file opened on specified unit; advances pointer to next sequential record.
POSITION	Moves the file pointer to any record in the file. Records are positioned to by number.
READ[*],READ LINE	Reads values from record to which pointer is currently positioned; advances pointer to the next record unless * is specified. Random reads can be done by POSITIONing the file pointer.
REWIND	Returns pointer to first record (top) of file.
ON END	Establishes action to be taken when pointer reaches the end of the file.
CLOSE	Closes file to reading and writing and releases file unit for other use.

Defining DAM Files

Direct access files are opened in the same manner as are SAM files. (See the DEFINE statement, above.) Records in a DA file can be set to a value larger or smaller than the default of 6Ø words (12Ø characters) when the file is first defined. The minimum is 4 words: the maximum 512. The record size of a DAM file is fixed to the supplied value, if given, or to the default value. This value remains in effect for every record added to the file. Details on adjusting the record size of a DA file can be found in Appendix E.

Direct access files are either ASCII or binary, as are sequential files. Direct access files are identified in BASIC/VM by the type-codes ASCDA or BINDA.

The following statement defines or opens an ASCII direct access file with a record size of 35 words (70 characters):

```
DEFINE FILE #1='DIRECT', ASCDA, 35
```

Writing Data To DAM Files

Data are stored in ASCDA files just as they are in ASCSEP files, i.e., with comma delimiters. Commas are inserted as internal data markers in both types, so string values containing commas will be broken up. Semicolons, commas and colons used as delimiters in WRITE statements are ignored, as shown below:

```
10 DEFINE FILE #1 = 'ASCDA', ASCDA
20 DEFINE FILE #2 = 'BINDA', BINDA
30 READ A$,B,C,D
40 DATA 'TRIANGLE DIMENSIONS',12,13,14
50 WRITE #1,A$
60 WRITE #1,B;C;D
70 WRITE #2, A$
80 WRITE #2,B,C,D
90 CLOSE #1,2
>RUNNH
STOP AT LINE 90
>TYPE ASCDA
TRIANGLE DIMENSIONS,

12,13,14,

>TYPE BINDA
<TRIANGLE DIMENSIONSF_1976
```

Random Access to DAM File Records

The major advantage of DAM files over SAM files lies in their record access flexibility. The file pointer can be moved to any record in the file with the POSITION statement. Records are positioned to by number. The number corresponds to the position of the record relative to the top of the file; i.e., record number one is at the top of the file, and so forth. The data in the currently positioned record can then be obtained with a READ statement.

The LIN#(unit) function can be used in DAM file access to check the actual record number to which the pointer is positioned. Instead of returning a line number as it does in ASCLN files, LIN #(unit) returns the number of the record in the file.

In the following example, a file is DEFINEd and data are added to it. The actual file contents are inspected by TYPEing the data file. Next the file is re-opened for READ access only and data are retrieved with POSITION and READ.

Example:

```
>10 DEFINE FILE #1 = 'ASCDA2', ASCDA,30
>20 WRITE #1, 'THIS IS THE INFORMATION FOR ROBERT SNORK'
>30 WRITE #1,'NAME: ROBERT SNORK; TITLE: OPTHAMOLOGIST'
>40 WRITE #1, 'ADDRESS: 12 MYOPIC LN, OCULAR, WISCONSIN'
>50 CLOSE #1
>RUNNH
STOP AT LINE 50
>TYPE ASCDA
THIS IS THE INFORMATION FOR ROBERT SNORK,
NAME: ROBERT SNORK; TITLE: OPTHAMOLOGIST,
ADDRESS: 12 MYOPIC LN, OCULAR, WISCONSIN,
>!
>!Open a READ only file
>DEFINE READ FILE #1 = 'ASCDA2', ASCDA
>POSITION #1 TO 3
>PRINT LIN#(1)
3
>READ #1,C$
>PRINT C$
ADDRESS: 12 MYOPIC LN
>PRINT LIN#(1)
4
>READ #1,A$
END OF FILE  AT LINE  0
```

In this example, the record size was set to 30 words (60 characters). In direct access, the file pointer scans through the file until the beginning of the desired record has been reached. The pointer determines the actual location of this record by counting the number of characters it has to bypass in order to reach it. The pointer knows how many characters are in each record, so it counts, in this case 2 times 60 characters, or 120 characters, to reach the third record. The pointer will now be at character 121, which begins the third record. The LIN# function clearly indicates that the pointer is now at the third record. After the READ is done, the pointer is at record #4, as indicated by the LIN #(unit) function. Note that when opening a file for READing only, the record size need not be given; however if an incorrect record-size is specified, the error message, DA RECORD SIZE ERROR AT LINE 0 is returned.

Reading DAM Files

DAM file READs are done in the same manner as are SAM file READ, i.e., READ, READ* and READLINE work in direct access just as they do in sequential access. Each READ statement advances the file pointer to the next sequential record when the READ has been completed. READ* holds the pointer at the current record until all the values specified in the next READ statement are satisfied. In ASCDA files, READLINE returns all the values in a record as one datum, commas, semi-colons, etc, included.

If a file has been previously DEFINEd and written to, it can be opened and restricted to READ or APPEND access by using the READ or APPEND options of the DEFINE statement, as in the previous example.

Error Trapping in DAM FIles

END OF FILE errors, as well as other execution mishaps, can be trapped via the ON END and ON ERROR statements discussed earlier under SAM file handling. These statements are applicable to diract access files as well as sequential files.

CLOSEing a DAM File

Direct access files are closed in the same manner as are SAM files. It is a good practice to close all data files when access to them has been completed. This ensures against accidental truncation which could occur if you break out of BASICV via CTRL-P or BREAK. When this happens, all opened file units are left open. At PRIMOS command level, these file units can be closed with the CLOSE ALL command, discussed in Section 3.

SEGMENT DIRECTORIES

A segment directory is actually a list of numbered entries which contain the addresses of data files. These numbered entries, referenced by number only (no names) are called 'pointers' because they point to, or reference, data files. The files to which they point can be of any BASICV file-type. Accessing the data files listed under a segment directory requires an additional step in the DEFINE process.

Opening a Segment Directory Data File

The process of opening a data file contained in a segment directory is best illustrated by example. Suppose we have in our current directory a segment directory SEGA which contains several pointers to data files. To access a certain ASCII data file referenced by one of these pointers (in this case, File 3), the segment directory must first be opened on a file unit as follows:

```
DEFINE FILE #1='SEGA', SEGDIR
```

The segment directory SEGA is now opened on file unit 1. Next, we must choose the third entry, which contains the data file address:

```
POSITION #1 TO 3
```

The file pointer is now positioned to segment 3 and the address of the data file we want to open. To open this data file, another file unit must be assigned with a DEFINE FILE statement. A special convention is used to indicate that the data file being opened is part of the segment directory opened on the previous unit:

```
DEFINE FILE #2='(SD1)', ASC
```

(SD1) refers to the file unit # 1 on which SEGA, the segment directory (SD) has been opened. The ASCII data file is now opened for reading or writing on unit 2. Both file units must remain open as long as the data file is being used.

Deleting Data Files From Segment Directory

Data files in a segment directory can be deleted by using the REPLACE statement. The format of the statement is:

```
REPLACE #unit SEG x BY SEG y
```

The pointers x and y point to two files, (x) and (y) respectively, REPLACE deletes file (x), and moves pointer y into pointer x, leaving pointer y empty. The original (x) is gone, and the original (y) has been renamed (x). Refer to Figure 8-2 for a visual interpretation.

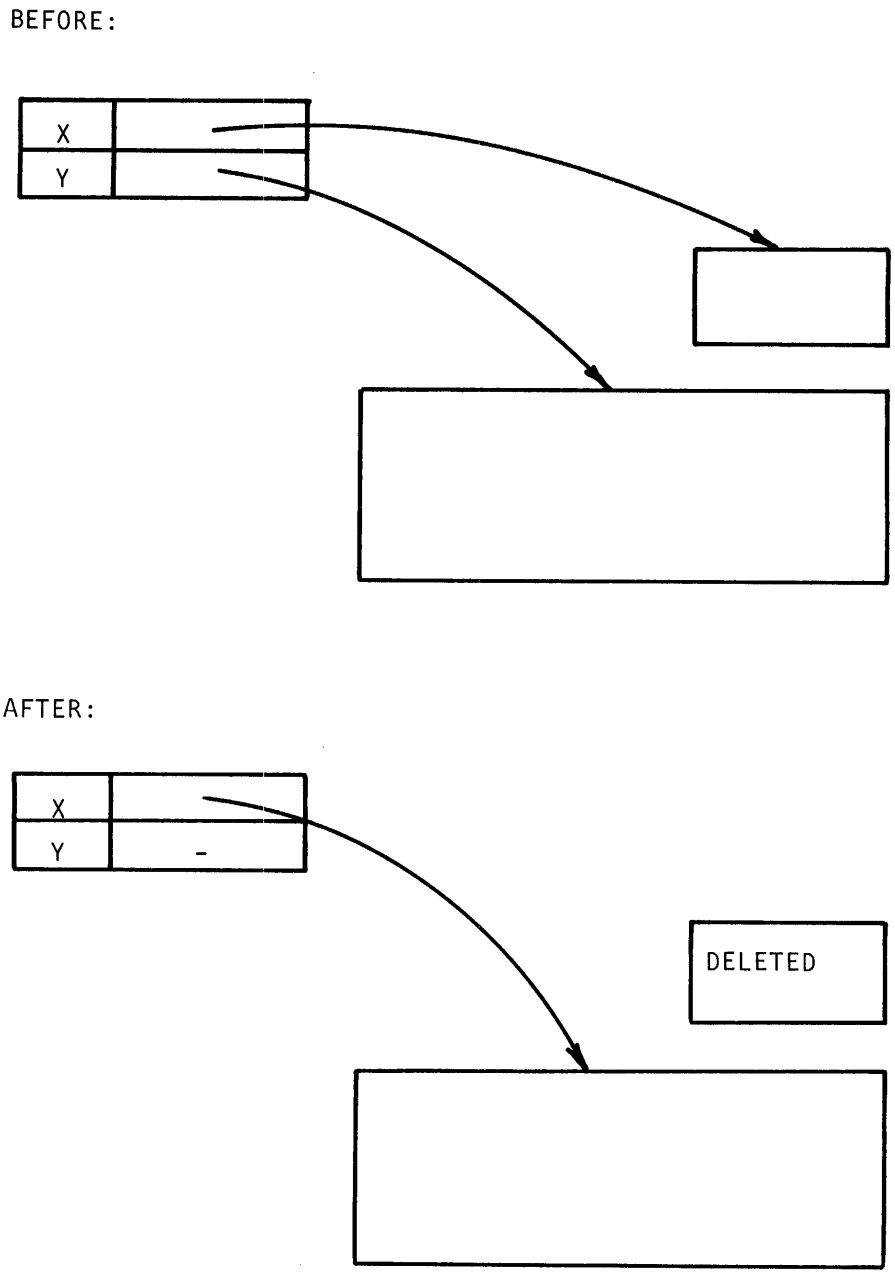


Figure 8-2. Deleting a SEGDIR Data File

MIDAS

MIDAS, or the Multiple Index Data Access System, is a collection of interactive utilities and subroutines for the efficient management of index-sequential and direct-access data files. MIDAS provides the programmer with an efficient method of building, restructuring, deleting, searching and accessing keyed-index data files. Data entry lockout protection and multiple user access to files are also supported by MIDAS. BASIC/VM does not support the following MIDAS features: direct-access and secondary-index data. For more information on the features of MIDAS and the usage of the related utilities, consult the REFERENCE GUIDE, MIDAS.

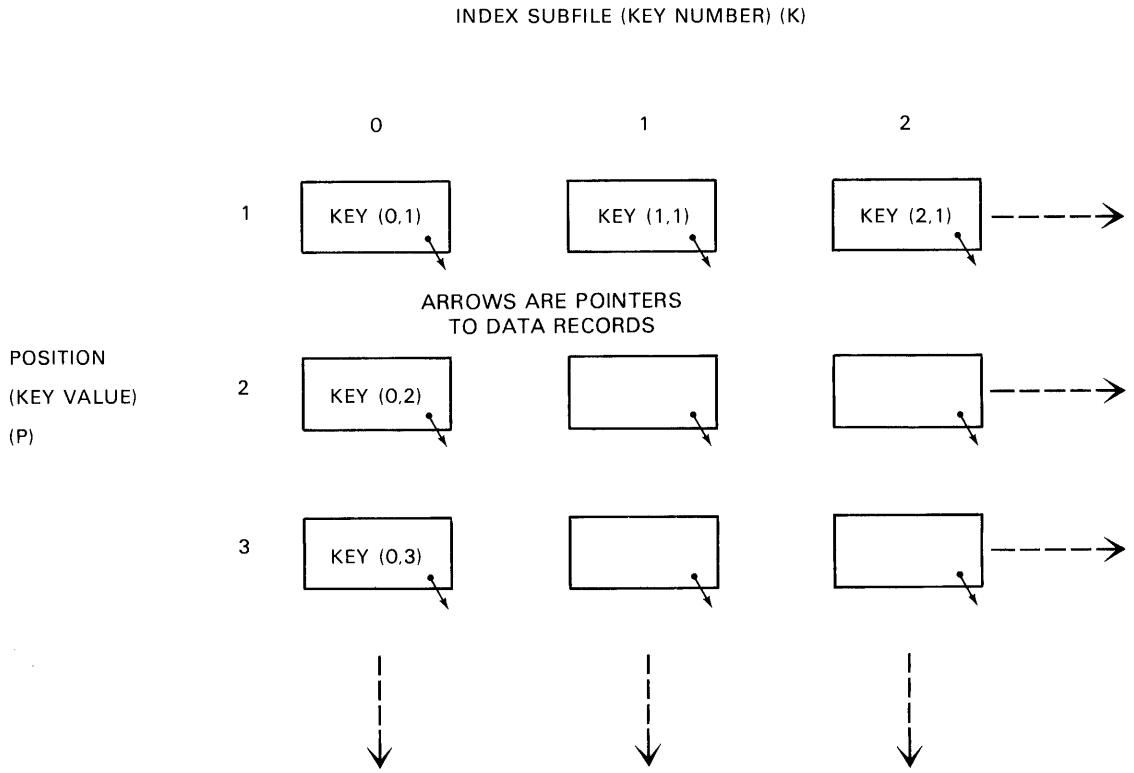
Brief Description of MIDAS Files

The first step in building a MIDAS file is the creation of a template, or file descriptor, for the MIDAS file. The PRIME-supplied program CREATK, is used to do this. It prompts the user for input describing the file to be created. The parameters supplied include the filename, access type (DA,SA), data subfile information including key type, key size for both primary and secondary indices. A MIDAS file can contain up to 2Ø indices, that is, 1 primary and 19 secondary indices.

Maintenance of the file can be done by multiple users simultaneously. A lockout subroutine guards against simultaneous changes and deletions of data entries. Other operations are done by exclusive single user access. See the MIDAS reference manual for further information.

MIDAS File Configuration

Although MIDAS provides its own methods of accessing files, the statements provided by BASIC/VM allow the user to access data in a MIDAS file and use it in a BASIC program. These statements can be thought of as operating on a MIDAS file configured as a rectangular matrix or two-dimensional array. Each element of the matrix contains a unique data record pointer. Access to these data records is accomplished by specifying the correct 'coordinates' of a particular element or key in the matrix. See Figure 8-3.



The MIDAS File modeled as a matrix consisting of index-subfile numbers (key numbers) on the X-axis, and key-positions (key values) on the Y-axis.

Figure 8-3. Configuration of MIDAS file

The values of K and P (in the previous diagram) form the coordinates of data record pointers.

During a file read, the 'READ' pointer moves around the array, allowing the user to obtain either the key or the data record pointed to by the key. Initially, the file 'READ' pointer is set to the first primary key or the upper left corner of the matrix.

STATEMENTS

The statements used to access MIDAS files in BASIC/VM are similar in function and format to those discussed earlier in this section. The parameters and arguments must be supplied in legal BASIC form. These statements are designed to perform a consistent and complete set of movements around a MIDAS file structure, so that any sequence of statements may be used without inconsistent or unpredictable results. Statements are listed below:

Statement	Function
DEFINE	Opens existing MIDAS file on specified unit.
ADD	Adds record to end of MIDAS file. Does not change current record location.
READ[KEY]	Reads data from MIDAS file: optional arguments specify record location. KEY option returns value of key to which pointer is currently positioned.
POSITION	Moves read pointer to any record in file; locks on record until pointer is re-positioned.
REWIND	Rewinds pointer to top of indicated column in file (see figure 8-3) or to beginning of file. (default).
UPDATE	Adds data to current record.
REMOVE	Deletes one or more keys from MIDAS file: if primary key, deletes associated data.
CLOSE	Closes MIDAS file on indicated unit.

TERMS AND CONVENTIONS

The following is a list of special terms and conventions used in defining the MIDAS access statements.

Definitions

{...} Select any one of the vertically stacked elements.

[...] Enclosed items are optional.

* Indicates repetition, Ø or more times.

+ Indicates repetition, 1 or more times.

Terms	Definitions
#unit	File unit on which MIDAS file is opened.
KEY Ø PRIMKEY	The primary key.
SAME KEY	Positions to or returns datum only if next key matches current one.
SEQ	Supplied in lieu of key: next sequential record is positioned and read.
num-expr-x	Represents numeric expressions.
str-expr-x	Represents string expressions.

MIDAS FILE HANDLING

The BASIC/VM statements available for MIDAS file access are described below. The examples supplied with the descriptions of each statement are taken from the MIDAS-BASIC demonstration program that immediately follows the presentation of statements.

Opening a MIDAS File

The DEFINE FILE statement opens a MIDAS file on an indicated file unit. If the record size is specified, the internal buffers are dimensioned to this value. The record size should be equal to the length of the data record. This information is defined in the MIDAS file by the CREATK utility.

```
DEFINE FILE #unit = str-expr, MIDAS [,num-expr-1]

   #unit      = file unit number on which file is opened
   str-expr   = MIDAS filename
   num-expr-1 = record size (in words)
```

Example:

```
DEFINE FILE #1= 'DIR', MIDAS, 64
```

Positioning the File Pointer

The position statement positions the read pointer to any record in the MIDAS file. The record positioned to is locked up on positioning and un-locked when the pointer is POSITIONed to another record. Note that there are no specific lock and unlock statements in BASIC/VM.

```
                                        SEQ
POSITION #unit, { KEY [num-expr] = str-expr }
                                      SAME KEY

   num-expr  =  secondary key number  (index subfile number)
   str-expr  =  key value(primary or secondary)
```

Example:

```
POSITION #1, SAME KEY
```

READing a MIDAS File

Data are retrieved from a MIDAS file via the READ statement. The KEY, SAME KEY or SEQ options are used to specify the location of the record to be read. The READ KEY statement gives the actual value of the current key to which the pointer is positioned.

```
                                          SEQ
READ[KEY] #unit [,{KEY [num-expr] = str-expr}], str-var
                                        SAME KEY

   num-expr = index subfile number
   str-expr = key value
   str-var  = variable into which data is read from record
```

Example:

```
READ #1, SEQ
```

If SEQ is used in place of a key, the next sequential record, in key order, is read. SAME KEY returns a datum only if the next key is the same as the current one. If the keys do not match, an error trap is taken. READ statements pre-position and lock to the location specified by the KEY, SAME KEY or SEQ (sequential)options. The data is then read and returned in the specified string variable. In the optionless form of READ, (e.g., READ #1, X$), no positioning occurs and only the current record is read.

Writing to a MIDAS File

The ADD statement adds a record to the MIDAS data base. It does not change the current record location.

```
                                  PRIMKEY
ADD #unit, str-expr-1, { KEY [Ø-expr] } = str-exp-2 keylist
where keylist = [,KEY num-exp-1 = str-expr-3] *
```

Ø-exp	=	expression evaluating to zero
str-expr-1	=	MIDAS filename
str-expr-2	=	actual primary key value
str-expr-3	=	value of secondary key
num-expr-1	=	secondary key number (index subfile number)
keylist	=	list of secondary key numbers and values

```
ADD #1, X$, KEYØ = I$(1)
```

The UPDATE statement adds a string expression to the current record on the specified file unit. If keys are being stored in the record, the UPDATE statement should not be used for changing these keys. BASICV does not monitor internal record structure and can not determine changes in a key field.

```
UPDATE #unit, X$
```

X$ = string expression to be written to current record.

Removing Data

The REMOVE statement deletes a given key from the MIDAS data base. If the key is a primary key (where num-expr=Ø) then the data associated with the primary key also is deleted. The language permits both multiple and single key removal in a single statement.

REMOVE #unit [, KEY [num-expr] = str-expr]+

Repositioning the File Pointer

The REWIND statement is used to reposition the file pointer from the current index subfile to a specified point. This can be thought of as positioning the pointer to the top of an indicated column. If the key specification is omitted, the default KEY Ø is assumed. This positions the pointer to the upper left corner of the matrix. (equivalent to REWIND #unit, KEY Ø.)

REWIND #unit [, KEY num-expr]

Example:

REWIND #1, KEY3 (Set pointer to top of index subfile [KEY]3)

Closing a MIDAS File

A MIDAS file is closed in the same manner as the data files previously discussed. The format is:

CLOSE #unit

SAMPLE PROGRAM

The following program, MIDASDEMO, illustrates the use of the BASIC/VM MIDAS access statements to retrieve record information from a MIDAS file. This program is included on the Master Disk shipped at Rev. 16 and is available for general use. A description of the functions defined in the program follows:

```
 1 !    ** A VERY SIMPLE 'MIDAS QUERY LANGUAGE' **
 2 !
 3 !        MIDAS Demonstration Program
 4 !
 5 !    This program demonstrates the use of MIDAS in a simple
 6 !    application. Central ideas to note are the use of multiple
 7 !    keys, storage of key fields as data, and the use of BASICV's
 8 !    string functions to automatically control string lengths,
 9 !    to perform space-padding, and facilitate string comparisons.
1Ø !
11 !    The functions available via this program are:
12 !       FIND [ALL] field-name field-value
13 !         Finds one or all of the records with a the given value
```

```
14 !       in the field specified by field-name. Field names
15 !       are requested from the user at the start of the program.
16 !      ADD
17 !       Allows the user to add a record to the data base.
18 !       The user is prompted with the field names before
19 !       being required to type in the record.
20 !      LIST
21 !       Lists out all records in the file.
22 !
23 !
100 ON ERROR GOTO 680  ! first set a single error handler
110 DIM I$(10)  ! the input array
115 !
116 !  First define all needed functions
117 !
120 DEF FNP$(X$,N)  ! pads X$ with spaces on right such that total
                      length is N
130  Y$=X$
140  Y$=Y$+' ' UNTIL LEN(Y$)=N
150  FNP$=Y$
160 FNEND
161 !
162 !
170 DEF FNK(F$)  ! returns a key (index subfile) number given a field
                   name
180   FOR I = 1 TO 10
190     FNK = I-1
200     IF K$(I)=F$ THEN GOTO 220
210   NEXT I
220 FNEND
221 !
222 !
230 DEF FNI  ! input function - gets space-separated strings from TTY
               and
231          ! stores the sequence in I$(1)...I$(n)
240   INPUTLINE '.',X$ ! prompt with a '.'
250   X$=X$+' '
260   MAT I$=NULL
270   FOR I = 1 STEP 1 UNTIL CVT$$(X$,2)='' ! CVT$$ insures no blanks
280     I$(I) = LEFT(X$,INDEX(X$,' ')-1)
290     X$ = RIGHT(X$,INDEX(X$,' ')+1)
300   NEXT I
310 FNEND
311 !
312 !
320 DEFINE FILE #1='DIR',MIDAS,64
330 MATINPUT 'Fields:',K$(*) ! field names, in order from KEY 0
331 !
332 !  ** main loop **
333 !
340 D=FNI ! input command string
345 !
346 !   FIND ALL
```

```
347 !
350 IF I$(1)='FIND' AND I$(2)='ALL' THEN DO
360   POSITION #1, KEY FNK(I$(3))=I$(4)
370   READ #1, X$
380   PRINT CVT$$(X$,16)    ! compress strings of blanks to one blank
390   POSITION #1, SAME KEY  ! find all records with this key value
400   GOTO 370
410   DOEND
411 !
412 !   FIND
413 !
420 IF I$(1)='FIND' THEN DO
430   READ #1, KEY FNK(I$(2))=I$(3), X$
440   PRINT CVT$$(X$,16)
450   GOTO 340
460   DOEND
461 !
462 !   ADD
463 !
470 IF I$(1)='ADD' THEN DO
480   PRINT K$(I): FOR I = 1 TO 4
490   PRINT ' ';
500   D = FNI
510   I$(1)=FNP$(I$(1),32)  ! write data must be padded to correct
                              length
520   I$(2)=FNP$(I$(2),32)
530   I$(3)=FNP$(I$(3),32)
540   I$(4)=FNP$(I$(4),32)
550   Z$=I$(1)+I$(2)+I$(3)+I$(4)
560   ADD #1,Z$,KEY0=I$(1),KEY1=I$(2),KEY2=I$(3),KEY3=I$(4)
570   GOTO 340
580   DOEND
581 !
582 !  LIST
583 !
590 IF I$(1)='LIST' THEN DO
600   REWIND #1 ! default is KEY 0
610   READ #1, X$
620   PRINT CVT$$(X$,16)
630   POSITION #1, SEQ
640   GOTO 610
650   DOEND
651 !
652 !
660 PRINT '?'  ! command error
670 GOTO 340
671 !
680!  a single error handler  !!!!
681 !
690 IF ERR=56 AND ERL=390 THEN GOTO 340
695 IF ERR=56 AND ERL=630 THEN GOTO 340
700 PRINT ERR$(ERR):'AT LINE':ERL  ! fall through to system error
720 END
```

Setting Up A MIDAS File

The MIDASDEMO program operates on a MIDAS file whose template is set up by the CREATK utility. A command file, listed below, invokes the CREATK utility and prompts the user for the template information. The template is the skeleton for the file. It defines the keys on which data will be searched.

After the template has been set up, the program can be run and the record information can be added to the file.

```
OK, * FIRST MAKE AN EMPTY MIDAS FILE
OK, CO C CREATK
OK, CREATK
GO
MINIMUM OPTIONS? YES

FILE NAME? DIR
NEW FILE? YES
DIRECT ACCESS? NO

DATA SUBFILE QUESTIONS

KEY TYPE: A
KEY SIZE = : W 16
DATA SIZE = : 48

SECONDARY INDEX

INDEX NO.? 1

DUPLICATE KEYS PERMITTED? YES
KEY TYPE: A
KEY SIZE = : W 16
USER DATA SIZE = :  (CR)


INDEX NO.? 2

DUPLICATE KEYS PERMITTED? YES
KEY TYPE: A
KEY SIZE = : W 16
USER DATA SIZE = :  (CR)


INDEX NO.? 3

DUPLICATE KEYS PERMITTED? YES
KEY TYPE: A
KEY SIZE = : W 16
USER DATA SIZE = :  (CR)
```

```
INDEX NO.? (CR)

OK, CO TTY
OK,
```

Description of MIDAS Demo Program

The MIDAS Demonstration program sets up a series of functions to make record access more flexible. Most of the MIDAS access statements described earlier are included in this program.

The user-defined functions in this program are:

FNP$(X$,N) - pads a given string, X$, with spaces to make it equal to length N. Uses system function LEN(Y$) which returns the number of characters of string Y$.

FNK(F$) - returns a key (index subfile) number given a field name.

FNI - the input function; accepts string input from the terminal (TTY) and stores it in an array. Uses the system functions: CVT$$ which reformats a given string according to the indicated mask (listed in table 1Ø-3); INDEX(X$,Y$)- computes the starting position of Y$ in X$.(In this case, finds first blank space in X$.) LEFT(X$,Y)- returns the left-most Y characters of string X$.(In this case, returns first characters immediately to the left of the first blank.) RIGHT(X$,Y) -returns right-most Y characters of string X$. In this case, those beginning after the first blank found in X$.

After the functions that accept and organize input for the data file have been defined, the program opens the MIDAS file called 'DIR' on file unit #1. A record size of 64 words is specified, meaning that no data in excess of 64 words will fit into one record.

```
DEFINE FILE #1 = 'DIR',MIDAS,64
```

The user is then prompted to input field names in order, as shown in the sample dialogue below. The function FNI forms an array of these input strings. The prompt character for user input is defined as a single dot.

The program then defines what will occur when the user inputs the words, FIND or FIND ALL. The FIND ALL function incorporates the BASICV statements POSITION and READ.

```
POSITION #1, KEY FNK(I$(3)) = I$(4)
```

The POSITION statement tells the read pointer to find the record referenced by the key(index subfile number), in this case FNK(I$(3)), whose value is given by the expression I$(4).

After the record is read and printed out, the pointer is told to position to the next record referenced by the previously specified key. This accomodates the use of duplicate keys(i.e., having more than one record or entry referenced by a single key).

```
POSITION #1, SAME KEY
```

Once a record is positioned to, the data in it can be READ into a specified string variable. The READ statement places all the data in the current record into string X$x.

```
READ #1, X$
```

In this case, the specified key is given by FNK(I$(3))=I$(4), and the record associated with this key will be read. The function then prints out the record and loops until all records corresponding to the given key are read and returned.

The FIND function is similar to FIND ALL but only retrieves one specific record, the first one it encounters fitting the description given by the key.

POSITION is not necessary when READing a MIDAS file. The read pointer will automatically position to the proper record when a key value is supplied with a READ statement.

```
READ #1, KEY FNK(I$(2))= I$(3),X$
```

Here, the key number and value are supplied and the record is positioned to and read. The function then prints out the data in X$ and returns the user to the input function (FNI) at line 34Ø.

If the user types ADD at line 34Ø, the program jumps to line 47Ø which begins an 'ADD' sequence. Data can then be added to the MIDAS data base with the ADD statement. First, the item must be padded to the correct length, which is accomplished by FNP$. This key information is then added to 'DIR' with the ADD statement.

```
ADD #1, Z$, KEY Ø= I$(1), KEY1=I$(2),KEY2=I$(3),KEY3=I$(4)

   (The subset containing 'KEY1' through'I$(4)' is a KEYLIST.)
```

The LIST function makes use of the MIDAS access statements REWIND,READ and POSITION to generate a listing of all the records in the MIDAS file. The pointer is first wound to the beginning or (upper left corner of the matrix) of the file:

```
REWIND #1
```

No key is specified, therefore, KEY Ø is assumed. The program reads the first record pointed to and prints it out. The next record is then positioned to with the statement:

```
POSITION#1, SEQ
```

The SEQ parameter tells the pointer to position to the next sequential record in the file. It may have the same key value as the record just read. This happens when duplicate keys are being used. This sequential 'position-read' routine is done until the end of the file is reached.

The remainder of the program handles errors, using the BASIC/VM error functions, ERR$ and ERR. See Section 15 for details.

The following is an example of an actual interactive terminal session during which the DEMO program was run.

```
NEW OR OLD: OLD DEMO
>RUN
DEMO      FRI, SEP Ø1 1978          17:46:24


Fields: NUM,NAME,CITY,STATE
.ADD
NUM NAME CITY STATE  .1 JONES BOSTON MASS
.ADD
NUM NAME CITY STATE  .2 JAMES NEWTON MASS
.ADD
NUM NAME CITY STATE  .3 SMITH NYC NY
.ADD
NUM NAME CITY STATE  .4 AMES ORANGE NJ
.LIST
1 JONES BOSTON MASS
2 JAMES NEWTON MASS
3 SMITH NYC NY
4 AMES ORANGE NJ
.FIND NAME JAMES
2 JAMES NEWTON MASS
.FIND ALL STATE MASS
1 JONES BOSTON MASS
2 JAMES NEWTON MASS
.FIND ALL NAME J     (partial key access - finds all names starting with
 'J')
2 JAMES NEWTON MASS
1 JONES BOSTON MASS
.FIND ALL STATE N
4 AMES ORANGE NJ
3 SMITH NYC NY
.controlC.
END OF DATA AT LINE 24Ø
>QUIT
```

DEMO RUN

Running the MIDASDEMO

The MIDAS file template set up by CREATK contains no data; they are entered by the user when the DEMO program is run. The first data requested by the program are the field names, which correspond to the primary and secondary keys. Data items are entered in response to the '.' character, which was established by the DEMO program as the input prompt.

In the example session on the previous page, the 'ADD' function is used to add four separate records to the data base. The entries NUM, NAME,CITY and STATE correspond to the primary key (KEY Ø), and secondary keys, KEY 1-3, respectively. The various program-defined functions discussed earlier are then utilized.

The Control-C break-out at the end of the program is possible only during input mode, or when the program is waiting for input from the terminal.

A complete summary of all the BASIC/VM MIDAS access statements can be found in reference Section 15 in the rear of this manual.

SECTION 9

ARRAYS AND MATRICES

INTRODUCTION

Arrays and matrices are one or two-dimensional tables of contiguous numeric or string values. A matrix consists of those elements in an array with non-zero subscripts. Each array or matrix element is represented by a subscripted value (integral only). In a two-dimensional array element representation, the first subscript represents rows, the second, columns. Thus, in array A, below, element (2,2) is in row 2, column 2. This is the last element in the array. Any non-integral subscript value entered is truncated to an integer before being used to locate the specified element.

The DIM statement is used to dimension an array or matrix by setting a limit on the number of elements it contains. For example, the statement DIM A(2,2) sets up a two-dimensional array with the following elements:

```
(Ø,Ø) (Ø,1) (Ø,2)
(1,Ø) (1,1) (1,2)     (1,1) (1,2)
(2,Ø) (2,1) (2,2)     (2,1) (2,2)

  ARRAY A                MATRIX A
                  (all subscripts non-zero)
```

Matrix A consists of those elements of array A having non-zero subscripts. The dimensions of matrix A are 2 by 2, i.e., two rows by two columns. The actual dimensions of array A are 3 by 3: three rows by three columns.

ARRAYS

Numeric Arrays

A numeric array name is a simple numeric variable. An array name, followed by one or two parenthesized values, indicates an element in the array. For example, A(5) and B9(6) are elements in one-dimensional arrays. The values of all elements in a numeric array are initialized to Ø at the beginning of the program in which they are defined. Numeric array elements are assigned values like any other numeric variable, e.g., B9(6)=2.

String Arrays

A string array is named by a simple string variable. String array elements are represented by an array name followed by one or two values enclosed in parentheses. For example, the following are elements in string arrays:

```
A$(5)          (one-dimensional array element)
B$(I+1,3)      (two-dimensional array element)
```

All elements of a string array are variable length character strings. Each element of a string array is initialized to null by the compiler at the beginning of program execution. String array elements are assigned values like all string variables, e.g., A$(2)='steve'.

Declaring an Array

Array dimensions are established either by the DIM statement, by a MAT statement (such as MAT PRINT) which references the array, or by the default value of (10) or (10,10). The value of any subscript must be within the range of the defined array dimensions.

The first element in any array is represented as (0) for a one-dimensional array or (0,0) for a two-dimensional array. However, these elements are not printed if MAT PRINT is used to output the array; only the matrix portion of the array will be output.

Example:

```
DIM A (5)
```

defines a one-dimensional array of six elements: A(0) through A(5). DIM statements may appear anywhere in the program. Before execution begins, BASIC/VM sets up the arrays internally using the following rules:

1. If an array element is referenced in a program, such as A(1), or A(2,3) but the array A has not been defined in a DIM statement, it is implicitly dimensioned to (10) or (10,10).

2. If an array is defined more than once by DIM, the first dimension sets its size.

The following statement defines a 10 by 10 array where element (1,1) has a value of 2. All other elements are 0. Its dimensions are 10 by 10 (default) because it has not been previously dimensioned by DIM.

```
A(1,1) = 2
```

Below are examples of one-dimensional and two-dimensional numeric arrays defined by DIM.

One-dimensional:

```
10 DIM A(8)
20 FOR N = 0 TO 8
30 A(N) = N
40 PRINT A (N)
50 NEXT N
> RUN
    0
    1
    2
    3
    4
    5
    6
    7
    8
>
```

Two-dimensional: (note that line 80 outputs matrix M, not array M)

```
 10 DIM M (3,4)
 20 FOR I = 0 TO 3
 30 FOR J = 0 TO 4
 40 M(I,J) = 3 * I - J+1
 50 NEXT J
 60 NEXT  I
 70 PRINT 'M'
 80 MAT PRINT M
 90 PRINT LIN (2); 'M'
100 FOR F = 0 TO 3
110 FOR G = 0 TO 4
120 PRINT M (F,G)
130 NEXT G
140 NEXT F
150 PRINT LIN (2); 'DONE'
>COMPILE
>EXECUTE
```

```
M
3                2                1                Ø
6                5                4                3
9                8                7                6


M
1
Ø
-1
-2
-3
4
3
2
1
Ø
7
6
5
4
3
1Ø
9
8
7
6


DONE
```

String Array Example:

```
     1Ø DIM B$(2,2)
     2Ø B$(1,1)='MICHELLE'
     3Ø B$(1,2)='BECKY'
     4Ø B$(2,1)='KAREN'
     5Ø B$(2,2)='ARLENE'
     55 MAT PRINT B$
     6Ø END
     >RUNNH
     MICHELLE            BECKY
     KAREN               ARLENE
```

MATRICES

Matrices are dimensioned by the DIM statement, as are arrays, or are automatically defined when referenced by a MAT (matrix) statement. Such a matrix is assigned default dimensions of (1Ø) or (1Ø,1Ø). The dimensions of the matrix may be changed to larger or smaller than original, using the MAT statement followed by new subscript values for the parameters (dim-1, dim-2). See Table 9-1.

MATRIX OPERATIONS

Matrix operations are valid only for that part of an array defined as a matrix, i.e., that portion with non-zero subscripts. Matrix operations include initialization, redimensioning, addition, subtraction, multiplication, inversion and transposition. All matrix operations begin with the keyword MAT and are listed in Table 9-1. All of the indicated operations, except MAT=NULL, can be performed on numeric matrices. String matrices can only be initialized to NULL or redimensioned with the (dim) option of the MAT statement.

Table of Matrix Operations

The following table lists all available BASIC/VM matrix operations. The parameter dim, e.g., (dim-1 [,dim-2]) represents a numeric constant or expression defining the dimensions of a matrix; num-expr represents a numeric expression by which a matrix may be multiplied.

Table 9-1. Matrix Operations

Type	Statements Used
Mathematical:	
All elements are initialized to zero. (Matrix may be redimensioned.)	MAT X = ZER [(dim-1 [,dim-2])]
All elements are initialized to one. (Matrix may be redimensioned.	MAT X = CON [dim-1 [,dim-2])]
All elements are initialized to zero except the main diagonal (elements with equal subscripts) which is all ones: identity matrix. (Matrix may be redimensioned.)	MAT X = IDN [(dim-1 [,dim-2])]
All elements of string array are nulled. Matrix is optionally redimensioned.	MAT A$ = NULL [(dim-1 [,dim-2])]
All elements of two matrices are added to or subtracted from each other.	MAT X = X + Y or MAT X = X - Y
All elements of a matrix are multiplied by an expression.	MAT A = (num-expr)*X
Two matrices are multiplied.	MAT A = X*Y
All elements are transposed.	MAT X = TRN(Y)
The square matrix is inverted.	MAT X = INV(Y)
Input/Output:	
Within the program.	MAT READ, CHANGE

Between program and terminal.	MAT INPUT, MAT PRINT
Between program and external files or devices.	MAT READ #, MAT WRITE #

Initializing Matrices

Matrices are assigned initial values by being equated to one of four matrix constants in a MAT statement. The matrix constants are identified by the mnemonics ZER, CON, IDN and NULL. The NULL constant applies to string matrices only; the remaining constants are used only for numeric matrices.

The constant ZER initializes each element of the specified matrix to Ø. The following statements define a 5 by 7 matrix and initialize each element to Ø, respectively:

```
DIM A(5,7)
MAT A = ZER
```

The constant CON initializes each element of the specified matrix to 1. The following statements define a 2 by 3 matrix and initialize each element to 1, respectively:

```
DIM B(2,3)
MAT B = CON
```

The constant IDN initializes the matrix to the identity matrix. All elements except those on the main diagonal are Ø. The diagonal elements are 1. For IDN to be valid, the matrix must be square.

Example:

```
DIM A(3,3)
MAT A = IDN
```

results in:

```
1 Ø Ø
Ø 1 Ø
Ø Ø 1
```

The constant NULL has the same effect on string arrays as ZER has on numeric arrays; it initializes each element of the matrix to a null value. The matrix can then be redimensioned.

Example:

```
>DIM A$(5)
>A$(I) = 'ABCD' FOR I = 1 TO 5
>MAT PRINT A$:
>ABCD ABCD ABCD ABCD ABCD
>MAT A$ = NULL
>MAT PRINT A$:
```

Redimensioning Matrices

The constants ZER, CON, NULL and IDN can also be used to change the dimensions of the matrix. By specifying subscripts after the constant, the matrix is redimensioned and the value of each element within the matrix is set according to the constant used. The following examples illustrate this concept using CON and ZER:

Example 1:

```
DIM A(4,5)
MAT A = CON(3,3)
```

Changes the dimensions of A from (4,5) to (3,3) and sets the value of each element to 1:

```
1 1 1
1 1 1
1 1 1
```

Example 2:

```
DIM X(3,3)
MAT X = ZER(4,2)
```

Changes the dimensions of X from (3,3) to (4,2) and sets the value of each element to Ø:

```
Ø Ø
Ø Ø
Ø Ø
Ø Ø
```

The dimensions of a matrix can also be changed by assigning to it a matrix of other dimensions.

Example:

```
DIM A(6,6)   !Total of 36 elements in matrix
DIM B(5,4)    !Total of 2Ø elements in matrix
MAT A = B     !A is a 5 by 4 matrix of 2Ø elements
```

Matrix Addition

The elements of two numeric matrices may be added and the values assigned to the corresponding elements of a third matrix. The example below adds the elements of matrix B to those of matrix C and stores the results in matrix A (called the target matrix):

```
MAT A = B + C
```

The source matrices (B and C) must have the same dimensions. The target matrix (A) is converted to the same dimensions as B and C.

Figure 9-1 diagrams the addition of two matrices to produce values in the corresponding elements of a third matrix.

One-dimensional Matrices:

```
MAT C = A   +   B

24          5       19
37          7       3Ø
34         12       22
```

Two-dimensional Matrices:

```
MAT C     =    A          +    B

26 22  6       14 2Ø  3        12  2  3
2Ø 18 26       19  4  7         1 14 19
45 28 13        8 23 1Ø        37  5  3
```

Figure 9-1. Matrix Addition

Matrix Subtraction

The values in the elements of two matrices may be subtracted to produce the values in corresponding elements of a third matrix as shown in the following expression:

```
MAT A = B - C
```

The elements of matrix A are set to the difference of the corresponding elements of matrix B and matrix C. Source matrices B and C must have the same dimensions. The dimensions of A become the same as those of B and C.

Matrix Multiplication

A matrix can be multiplied by a numeric expression or by another matrix. There are certain restrictions however, as explained below.

Scalar Multiplication: A matrix may be multiplied by a numeric scalar expression and the results stored in a second or target matrix.

In the following example, each element of matrix Y is multiplied by 5 and the resulting values are assigned to matrix A. The dimensions of matrix Y then become those of matrix A.

```
MAT A = (5)*Y
```

Multiplication of Two Matrices: To multiply two matrices, both must be two-dimensional and the number of columns of the first matrix must equal the number of rows of the second matrix. The result is a third matrix with the same number of rows as the first matrix, and the same number of columns as the second matrix.

The example below multiplies matrix B and matrix C to produce matrix A with dimensions of 2 by 4.

```
DIM B(2,3)
DIM C(3,4)
MAT A = B*C   !A is a 2 by 4 matrix
```

Each element in matrix A is the result of multiplying the elements of matrix B times the elements of matrix C in the following way:

1. Multiply each element in row 1 of B, by each element in column 1 of C.

2. Add the results to obtain the value of the element in matrix A, row 1, column 1.

3. Continue the pattern by multiplying row 1 and column 2 to produce element A(1,2); row 1 and column 3 to produce A(1,3); row 2 and column 1 to produce A(2,1); row 2 and column 2 to produce A(2,2); and row 2 and column 3 to produce A(2,3).

Example:

```
 B       C

1  2    7   8   9
3  4   10  11  12
5  6   16  17  18


   A = B * C
   A(1,1) = (1*7) + (3*8) + (5*9)
   A(1,1) = 7 + 24 + 45

   The first element in matrix A = 76
```

The current values of the target matrix may not be used as part of the multiplication expression:

```
MAT A = A*B (illegal)
```

Inverting and Transposing Matrices

Matrix inversion is accomplished by using the INV function. A matrix can be inverted only if it is square and its determinant is not zero. Multiplying a matrix by its inverse yields the identity matrix. The determinant of a matrix is determined by the DET function (a numeric system function - see Section 1Ø). More information on 'DET' and matrix operations can be found in the following reference, or in most Linear Algebra texts.

Thomas, George B., Calculus and Analytic Geometry, 4th edition, Addison-Wesley, Reading, Mass., 1968

Zuckerberg, Hyam L., Linear Algebra, Charles E. Merrill, Pub., Columbus, Ohio, 1972

The following example demonstrates the use of the inverse and determinant functions:

```
MAT READ A
IF DET A = Ø  THEN PRINT 'CANT INVERT' ELSE PRINT 'CAN INVERT'
IF DET A <> Ø  THEN MAT C= B*INV(A)
DATA 1,2,3,1
```

It is often necessary to transpose a matrix so that it may be multiplied by another matrix. (Remember that two matrices can be multiplied only if the number of columns of the first matrix equals the number of rows of the second matrix.) For example, if a company sells four products and has three customers buying varying amounts of each product, the quantities would be set up in a 3 by 4 matrix (A). The cost of each product would be set up in a 1 by 4 matrix (B). To determine the amount owed by each customer, matrix A must be multiplied by matrix B. Since you cannot multiply (3,4) * (1,4), you must transpose B:

```
C=A*TRN(B) or C=(3,4)*(4,1)
```

Figure 9-2 illustrates this concept.

Matrix A

	nuts	bolts	nails	screws
Customer 1	100	100	60	0
Customer 2	0	0	300	10
Customer 3	50	100	100	100

Matrix B

	nuts	bolts	nails	screws
Cost/item	.10	.05	.01	.02

C = A * TRN(B)
=(3x4)*(4x1)
=(3x1)

Matrix C

Customer 1 Owes	Customer 2 Owes	Customer 3 Owes
15.60	3.20	13.00

Figure 9-2. Matrix Inverse

Every two-dimensional matrix has a transpose. This is determined by rolling the matrix on the main diagonal. If matrix A is:

```
4 2
3 1
```

its transpose is:

```
4 3
2 1
```

and can be assigned to another matrix with the statement:

```
MAT B = TRN(A)
```

Both matrices are two-dimensional. The dimensions of matrix B are set to the reverse of those of matrix A. For example, if MAT A is (2,3), MAT B will become (3,2). It is not legal to assign the transpose of a matrix to itself, i.e., MAT A = TRN(A) is illegal.

Transmitting Matrix Data Within A Program

As with general I/O, data can be read via MAT READ from DATA statements within the program. The values are assigned to each element of the matrix specified in MAT READ.

The dimensions of the matrix are first defined in a DIM statement. Then data values are read from DATA statement(s) until each element of the matrix has a value. The following example reads fifteen numbers from consecutive DATA statements and assigns them to matrix A:

```
DIM A(3,5)
MAT A = ZER
MAT READ A
DATA 1,2,3,4,5,6,7,8,9
DATA 1Ø,11,12,13,14,15
```

The result is:

```
A(1,1) = 1
A(2,1) = 2
  .
  .
  .
A(3,5) = 15
```

Data Conversion

Data may also be changed from a string of ASCII characters to a one-dimensional array containing the decimal equivalents of the characters (including parity). Conversely, a decimal array may be changed to its corresponding string of ASCII characters. Refer to Appendix B for the ASCII character table. To do either, use the CHANGE statement.

The following program changes the ASCII characters in D$ to an array of their decimal equivalents:

```
10 DIM A(6)
20 D$ = 'CHANGE'
30 CHANGE D$ TO A
40 FOR I = 1 TO 6
50 PRINT A(I)
60 NEXT I
```

When run, the program prints:

```
195
200
193
206
199
197
```

If the array has not been previously dimensioned, the CHANGE statement automatically dimensions the array to the length of the string. The zeroth element of the array contains the length of the string. In the above example, A(0)=6, because 'CHANGE' is six characters in length.

The following program converts decimal values listed in a DATA statement, into their corresponding ASCII characters and prints them:

```
10 FOR X = 1 TO 6
20 READ A(X)
30 NEXT X
40 DATA 208, 210, 197, 211, 212, 207
50 CHANGE A TO A$
60 PRINT A$
```

When run, the program prints:

```
PRESTO
```

Changing the zeroth element of a numeric array will alter the length of the corresponding character string of ASCII letters accordingly. For example, if A$='ABCDE' is converted to its numeric array equivalent, the first array element, A(Ø), will be equal to 5 because the string is five characters long. If the value of A(0) is changed to 3, a CHANGE of A to A$ will result in a string only three characters in length, e.g., A$='ABC'.

Transmitting Matrix Data Between a Program and the Terminal

Data to be stored in a matrix may be entered from the terminal. The MAT INPUT statement is identical to the INPUT statement except that the values entered are stored in matrix format.

The following program segment defines matrix B as having six elements. The MAT INPUT statement will expect six values to be entered. Each value will then be assigned to an element in matrix B:

```
DIM B(2,3)
MAT INPUT B
MAT PRINT B
```

When run, the following occurs: (user input is underlined)

```
!10
!15
!7
!20
!25
!13

 10            15            7
 20            25            13
```

The MAT PRINT statement prints the values of the entire matrix at the terminal.

The MAT INPUT mat (*) statement is a variant of the MAT INPUT statement. It accepts one line of input and automatically dimensions the matrix, named by mat, to the number of items input.

Matrix I/O to Data Files

Matrices can be written to and read from data files with the MAT READ #unit and MAT WRITE #unit statements. Only the file handling statements which deal with matrix I/O are presented here. Other data file handling operations are dealt with in Section 8 and Appendix E.

WRITEing a Matrix to a File

Matrices are written to a data file with the MAT WRITE statement. First, the matrix is defined with a DIM statement. Values are then assigned to each matrix element. The entire matrix is then written to the indicated data file with a single MAT WRITE statement.

Example:

```
>DEFINE FILE #1 = 'MAT'
>DIM A(3)
>A(1)=10
>A(2)=20
>A(3)=30
>MAT WRITE #1, A
>TYPE MAT
10                    20                    30
```

Reading From a File To a Matrix

Values written to a file with MAT WRITE can be read back with any of the READ statements covered in Section 8. They will be returned as any other data values stored in a file. The contents of a record or records in a file can be read into a matrix or matrices with the MAT READ or MAT READ* statements. The indicated matrices must first be defined with a DIM statement. Values are then retrieved from the record or records of the indicated file until the matrix or matrices are filled.

The following example illustrates the use of MAT READ and MAT WRITE. Notice that an error trap for an end of file is included for the default ASCII file. An END OF FILE message is generated when a MAT READ is attempted on an ASC file. Values from an ASC file can be read into a matrix by defining the matrix and reading record values into the indicated variable with a READ statement. MAT print can then be used to obtain a list of the values read into the indicated variable.

Example:

```
5 ON END #1 GOTO 120
10 DEFINE FILE #1 = 'MAT'
20 DEFINE FILE #2 = 'MAT2', ASCSEP
30 DEFINE FILE #3 = 'ASCDA', ASCDA
40 WRITE #1, 20,20,20
50 WRITE #2, 20,20,20
60 WRITE #3, 20,20,20
70 REWIND #1,2,3
80 DIM A(2)
90 PRINT 'READ MAT'
100 MAT READ #1, A
110 MAT PRINT A
120 MAT READ #2, A
130 PRINT 'READ MAT2'
```

```
140 PRINT
150 MAT PRINT A
160 MAT READ #3, A
170 PRINT 'READ ASCDA'
180 MAT PRINT A
190 REWIND #1,2,3
>RUNNH
READ MAT
READ MAT2

 20                  20

READ ASCDA
 20                  20

STOP AT LINE 190
>TYPE MAT
 20                  20                  20
>TYPE MAT2
 20, 20, 20,
>TYPE ASCDA
 20, 20, 20,
```

MAT READ* Statement

The MAT READ* statement functions just like the READ* statement. The read pointer remains positioned at the current record after a MAT READ* is executed rather than pre-positioning to the next record. This allows the next MAT READ* statement to continue reading data from the current record.

Example:

```
10 DEFINE FILE #1 = 'MATF', ASCSEP
20 WRITE#1, 10,10,10,10
30 WRITE #1, 20,20,20,20
40 WRITE #1,30,30,30,30
50 REWIND #1
55 DIM A(5)
60 MAT READ * #1,A
62 MAT PRINT A
65 MAT READ * #1, A
70 MAT PRINT A
75 REWIND #1
80 READ #1,A
85 MAT PRINT A
90 READ #1,A
95 MAT PRINT A
100 REWIND #1
105 MAT READ #1,A
110 PRINT A
115 PRINT
```

```
120 MAT PRINT A
130 REWIND #1
>MARGIN 20
>RUNNH
10
10
10
10
20

20
20
20
30
30

20
20
20
30
30

20
20
20
30
30

20

10
10
10
10
20

STOP AT LINE 130
>MARGIN OFF
>RUNNH
10                10                10                102
0
20                20                20                303
0
20                20                20                303
0
20                20                20                303
0
20

10                10                10                10
20
```

Notice the difference between MAT PRINT and PRINT. PRINT merely prints the value associated with a scalar variable. MAT PRINT outputs the matrix named by a particular variable, e.g., A. This distinguishes the variable A from matrix A, as shown in the example. Notice also that the MARGIN statement can be used to alter the length of lines printed at the terminal. This is useful for printing matrices in list, or tablular form.

SECTION 1Ø

NUMERIC AND STRING FUNCTIONS

INTRODUCTION

Most arithmetic operations can be simplified by using pre-defined numeric functions to handle routine calculations such as computing square roots. String data handling can also be facilitated by functions designed specifically for manipulating strings. BASIC/VM provides both numeric and string system functions to perform operations like calculating sine and cosine, generating random numbers, and converting a string to its corresponding numeric value. In addition to system provided functions, users can define their own functions to perform special routines within a program. The functions available in BASIC/VM are:

1. Numeric system functions
2. String system functions
3. Numeric and string user-defined functions:

This section lists all the currently defined system functions, (both numeric and string), and provides information on defining and implementing user-defined functions of both types. A detailed discussion of call-by-value and call-by-reference functions is also included.

NUMERIC SYSTEM FUNCTIONS

A numeric function is identified by a three or four letter name (such as TAN) followed by one or more parameters enclosed in parentheses. If more than one parameter is required, they are separated by commas. Numeric functions operate on numeric items or expressions. The result of a function operation is a single numeric value. Therefore, the function can be used anywhere in an expression where a numeric constant or variable can be used.

The following table lists the numeric system functions provided by BASIC/VM. In all of the descriptions, X represents any numeric expression, and Y and Z represent any integers.

Table 1Ø-1 Numeric System Functions

ABS(X)	Computes the absolute value of X.
ACS(X)	Computes the principal arccosine of X. The result is in radians in the range of Ø to PI. 36Ø degrees = 2 PI radians.
ASN(X)	Computes the principal arcsine of X. The result is in radians in the range of -PI/2 to PI/2.
ATN(X)	Computes the principal arctangent of X radians. The result is in the range of -PI/2 to PI/2.
COS(X)	Computes the cosine of X. The argument is in radians. The result is in the range -1 to +1.
COSH(X)	Computes the hyperbolic cosine of X (defined as (EXP(X)+EXP(-X))/2).
DEG(X)	Computes the number of degrees in X radians, [(18Ø/PI*X)]. The result is in degrees.
DET(X)	Computes the determinant of matrix X. If DET(X) unequal to Ø, matrix X has an inverse.
ENT(X)	Computes the greatest integer that is less than or equal to X.
ERL	Returns the statement number of the line which caused an error.
ERR	Returns the error code number of the last error.
EXP(X)	Computes e raised to the X power.
INT(X)	Performs integer truncation. If X=>Ø, returns the greatest integer <=X. If X<Ø, returns the least integer >-X.
LIN#(X)	For ASC LN files, returns the statement number stripped from the last input on unit X. For BIN DA files, returns the current record position of the file on unit X.
LOG(X)	Computes the natural logarithm (base e) of X.
NUM	Returns the actual number of entries to MAT INPUT M(*) and MAT INPUT M$(*) statements. Matrix M is one-dimensional.
PI	Computes the value of PI (3.14159).

RAD(X)	Computes the number of radians in X degrees.
RND(X)	If X>Ø, uses X to initialize the random number generator and returns X as the function value. If X<Ø, uses X to initialize the random number generator, and returns a value in the range zero to one. If X=Ø, returns a random number in the range zero <=result < 1.
SGN(X)	Computes a value based on the sign of X as follows: X<Ø SGN(X)=-1 X=Ø SGN(X)=Ø X>Ø SGN(X)=1
SIN(X)	Computes the sine of X. The argument is in radians. The result is in the range -1 to +1.
SINH(X)	Computes the hyperbolic sine of X defined as (EXP(X)-EXP(-X))/2.
SQR(X)	Computes the positive square root of X.
TAN(X)	Computes the tangent of X. The argument is in radians.
TANH(X)	Computes the hyperbolic tangent of X defined as (EXP(X)-EXP(-X)/EXP(X)+EXP(-X)).

Using Some System Functions

INT: The INT function can be used to round numbers:

```
INT (2.9 + .5) = INT (3.4) =  3.Ø
```

The INT function can also be used to round any numeric value to a specific number of decimal places.

```
INT  (1Ø * X1 + .5) /1Ø
```

rounds X1 to 1 decimal place.

```
INT  (1ØØ * X1 + .5) /1ØØ
```

rounds X1 to 2 decimal places.

RND: The RND function generates random numbers.

The following program yields twenty random integers of three digits or less:

```
1Ø  REM PROGRAM TO PRINT RANDOM NUMBERS OF 3 DIGITS OR LESS>
2Ø  FOR I=1 TO 2Ø
3Ø  L=RND(Ø)
35  L1=INT(L*1ØØØ)
4Ø  PRINT L1
5Ø  NEXT I
```

The RND function can also be used to define random numbers within a specific range. This is demonstrated by the following guessing game program.

```
 1Ø  PRINT 'WHAT NUMBER AM I THINKING OF'
 2Ø  FOR C=Ø TO 3
 3Ø  N=INT (5Ø * RND(Ø) + 1)
 4Ø  INPUT X
 5Ø  IF X<N GOTO 9Ø
 6Ø  IF X>N GOTO 11Ø
 7Ø  PRINT 'RIGHT ANOTHER'
 8Ø  GOTO 15Ø
 9Ø  PRINT 'TOO LOW'
1ØØ  GOTO 12Ø
11Ø  PRINT 'TOO HIGH'
12Ø  NEXT C
13Ø  PRINT 'TIME IS UP, ANOTHER'
14Ø  INPUT A$
15Ø  IF A$ = 'Y' GOTO 12Ø
```

The following program uses many of the system functions previously described.

```
100 ! EXAMPLE TO SHOW USE OF SYSTEM FUNCTIONS
110 !
120 ! DAH 12/13/77
130 !
140 LET V = RAD(1)            ! 0.01745329251994
150 ! 1 DEGREE IN RADIANS
160 W = RAD(30)               ! 0.5235987755983
170 X = RAD(45)               ! 0.7853981633974
180 Y = RAD(60)               ! 1.047197551197
190 Z = RAD(90)               ! 1.570796326795
200 ! W,X,Y,Z EQUIVALENTS 30,45,60,90 DEGREES RESPECTIVELY
210 !
220 ! TRIGONOMETRIC FUNCTIONS CALCULATIONS
230 S1 = SIN (V)
240 S2 = SIN (W)
250 S3 = SIN (X)
260 S4 = SIN (Y)
270 S5 = SIN (Z)
280 C1 = COS (V)
290 C2 = COS (W)
300 C3 = COS (X)
310 C4 = COS (Y)
320 C5 = COS (Z)
330 T1 = TAN (V)
340 T2 = TAN (W)
350 T3 = TAN (X)
360 T4 = TAN (Y)
370 T5 = TAN (Z)
380 A1 = ATN (T1)
390 A2 = ATN (T2)
400 A3 = ATN (T3)
410 A4 = ATN (T4)
420 A5 = ATN (T5)
430 PRINT 'DEGREES', 'SIN', 'COS', 'TAN', 'ARCTAN'
440 PRINT
450 PRINT 1, S1, C1, T1, A1
460 PRINT 30, S2, C2, T2, A2
470 PRINT 45, S3, C3, T3, A3
480 PRINT 60, S4, C4, T4, A4
490 PRINT 90, S5, C5, T5, A5
500 !
510 ! ARITHMETIC FUNCTIONS (LOG ETC)
520 !
530 X = 7.50
540 L = LOG(X)
550 E = EXP(X)
560 Q = SQR(X)
570 A = ABS(X)
580 I = INT(X)
590 P = SGN(X)
600 PRINT
610 PRINT
620 PRINT 'NUMBER =', X
```

```
630 PRINT 'LOG(X)=', L
640 PRINT 'EXP (X)=', E
650 PRINT 'SQUARE ROOT', Q
660 PRINT
670 PRINT 'ABS(X)', 'INT(X)', 'SIGN(X)'
680 PRINT
690 PRINT A, I, P
700 PRINT
710 PRINT
720 ! RANDOM NUMBER FUNCTIONS
730 !
740 PRINT 'RANDOM NUMBER FUNCTIONS'
750 PRINT
760 PRINT 'RND(0)', 'RND(N)', 'RND(-N)'
770 PRINT
780 Z1 = RND(0)
790 Z2 = RND(1)
800 Z3 = RND(-1)
810 PRINT Z1, Z2, Z3
```

Sample Output:

>DEGREES	SIN	COS	TAN	ARCTAN
1	.01745240643728	.9998476951563	.01745506492822	.01745329251994
30	.5	.8660254037844	.5773502691896	.5260879918124
45	.7071067811865	.7071067811866	.999999999999	.7878873796115
60	.8660254037844	.5	1.732050807569	1.047197551197
90	1	4.464471677451E-14	2.239906695009E+13	1.570796326795

NUMBER=	7.5
LOG (X)	2.014903620542
EXP	1808.042414456
SQUARE ROOT	2.738612787526

ABS (X)	INT(X)	SIGN(X)
7.5	7	1

RANDOM NUMBER FUNCTIONS

RND (0)	RND (N)	RND (-N)
.2112731933594	1	.2112731933594

STRING SYSTEM FUNCTIONS

String functions are used to obtain information about, or to operate on, a string or portions of a string. For example, the function SUB(X$,Y,[Z]), returns a substring, beginning with character Y, of a larger string, X$. String functions, e.g., STR$(X), can also convert a numeric item to its corresponding string representation; or they convert the string representation of a number to the numeric value it represents, e.g. VAL (X$). A string function is identified by a three to five letter name followed by one or more parameters enclosed in parentheses. Parameters can be numeric or string items depending on the type of operation the function performs.

Table 10-2 alphabetically lists the string functions provided by BASIC/VM. In all descriptions, X represents any numeric expression, Y and Z represent any integers, and X$ represents any string expression.

Table 10-2 String System Functions

CHAR(X)	Returns the character whose ASCII code is X. X is in the range 128-255.
CODE(X$)	Computes the decimal ASCII code of the first character of X$. Codes are listed in Appendix B. Note: the code of a null string is -1.
CVT$$(X$,Y)	Reformats X$ according to the mask Y. (Masks are listed in Table 10-3).
DATE$	Returns the date as YYMMDD.
INDEX(X$,Y$,[Z])	Computes the starting position of Y$ in X$, optionally beginning at character Z.
LEFT(X$,Y)	Returns leftmost Y characters of X$.
TIME$	Returns the time as HHMMSSFFF. (FFF is milliseconds)
MID(X$,Y,Z)	Returns Z characters of X$ starting at position Y.
LEN(X$)	Returns the length (number of characters) of string X$.
RIGHT(X$,Y)	Returns rightmost characters of X$ beginning with character number Y.
STR$(X)	Returns the string representation of the number X.
SUB(X$,Y,[Z])	Returns a substring composed of characters in positions Y through Z of string X$. If Z is not specified, the result is a one character substring consisting of character Y of string X$.
VAL(X$,[Y])	Converts a string, X$, to the numeric value it represents. Y will have the conversion status: 0=successful, 1=unsuccessful. If unsuccessful, run-time error occurs.

Table 1Ø-3. Masks For CVT$$

MASK	FUNCTION
1	force parity bit off
2	discard all spaces
4	discard .NUL. ,.NL. ,.FF. ,.CR. ,.ESC.
8	discard leading spaces
16	reduce multiple spaces to one space
32	convert lower case to upper
64	convert [to (and] to)
128	discard trailing spaces

(Masks can be combined additively)

Using String Functions

The following example utilizes several string system functions. Appendix A contains a sample program that uses string functions more extensively to format a block of text.

Example:

```
1Ø REM USING SOME STRING FUNCTIONS
2Ø PRINT
3Ø X$='SOMEBODY KILLED HER HUSBAND'
4Ø PRINT 'VALUE OF FOLLOWING STRING:'
5Ø PRINT
6Ø PRINT X$
7Ø PRINT
8Ø L1 =LEN(X$)
9Ø PRINT 'LENGTH OF STRING=': L1
11Ø PRINT
115 B$ = SUB(X$,21,28)
12Ø PRINT 'SUBSTRING IN POSITIONS 21 -28 IS:':B$
15Ø END
```

When run, the program results in the following output:

```
VALUE OF FOLLOWING STRING:

SOMEBODY KILLED HER HUSBAND

LENGTH OF STRING= 27

SUBSTRING IN POSITIONS 21 -28 IS: HUSBAND
```

USER DEFINED FUNCTIONS

In some programs it is necessary to execute the same sequence of statements or mathematical formulas in several different places. BASIC/VM allows you to define your own functions and use them just like system functions.

Numeric User-Defined Functions

The name of a user-defined numeric function consists of the letters FN followed by a letter or a letter and a digit as shown below:

```
FNA or FNA7
```

A reference to a user-defined function consists of the name of the function followed by a parenthesized argument expression. A function must be defined by a DEF statement. This definition must occur prior to the place where the function is called or referenced in the program.

Example:

```
DEF FNA (X2) = 3.14 * X2+2
```

A user-defined function reference may be included as an operand in an expression such as:

```
LET A1 = 3.14/FNA(X1)
```

The argument of a user-defined function may be an arithmetic expression. The expression is evaluated, and then the value of the expression is substituted for the argument in the function definition.

Example:

```
LET A1 = 3.14 * FNA (X1 + COS(B))
```

A user function may also be more than one line. After the last line of the function, use the FNEND statement to indicate the end of the function definition.

Example:

```
DEF FNA (I)
IF I=0 THEN FNA=-1 ELSE FNA = I*I+I
FNEND
```

When program execution begins, the function definition is ignored until the function is referenced. Each time it is referenced, program control returns to the lines which defined the function.

String User-Defined Functions

As with numeric functions, you may define your own string functions.

The name of the function consists of the letters FN followed by a letter (or a letter and a digit) and a dollar sign ($) as shown below:

FNA$ or FNA7$

A user-defined function must be defined by a DEF statement. If it is more than one line long, the last line must be FNEND, indicating termination of the definition. When program execution begins, the function definition is ignored until the function is referenced. Each time it is referenced, program control returns to the lines which defined the function.

Example:

```
10  REM A PROGRAM WITH SUBROUTINES
20  REM AND STRING FUNCTIONS
30  DEF FNA$ (X$, Y$)
40  FNA$=X$
50  IF X$>Y$ THEN FNA$=Y$
60  FNEND
70
 .
 .
100  IF A=1 THEN 1000
 .
 .
150  X$=FNA$(B$,C$)+D$
 .
 .
1000  PRINT 'A EQUALS ONE'
 .
 .
2000  GOSUB 5000
 .
 .
2500  B$=FNA$(X$,Y$)+B$
3000  GOTO 100
 .
 .
5000  Z$=FNA$(X$,Y$)+C$
5010  PRINT 'LINE 5010'
 .
 .
6000  RETURN
```

Program executions begins at line 70. Assuming A=1, control jumps from line 100 to line 1000. At line 2000, control transfers to line 5000. Because the function is referenced, control transfers to lines 30 through 60 where the function is defined. Control then returns to line 5010.

User-defined string functions may also be used in conjunction with system functions, as in:

```
DEF FNF$ (A,B,C) = LEFT (STR$ (A+B+C),5)
```

Defining Functions

A function in BASIC/VM is defined with a parameter, e.g., DEF FNA(x), where x is the parameter. The parameter must be an identifier for a variable or for an array. A parameter is sometimes refered to as a dummy variable.

A function is called in the program by an expression consisting of the function name (e.g., FNA) followed by a parenthesized argument or set of arguments.

Example:

```
DEF FNA(x)      x is a parameter of function FNA

x=5             dummy variable set equal to 5

FNEND           denotes end of function
  .
  .
  .
y=10            external program variable

Z=FNA(y)        expression calling function FNA; y is argument,
                sharing value with dummy variable x.
```

Call-By-Reference vs. Call-By-Value

Generally, there are two ways in which an argument can reference or set parameters in the function it calls: by value or by reference. This relationship between arguments and parameters determines whether the function is termed call-by-reference or call-by-value. This, in turn, is dependent upon the language in which the function is being used. In BASIC/VM, functions are call-by-reference.

In call-by-reference functions, arguments set parameters by reference. When a parameter is set by reference, every subsequent reference to that parameter actually becomes a reference to the storage location (slot where the value of the variable is stored in memory), of the argument that set it when the function was called. In other words, every assignment of a value to the parameter is in effect an assignment of the same value to the argument that set the parameter. (The

argument itself is essentially substituted for the parameter in the function.)

In the case of call-by-value, however, the value of the argument is actually copied to the storage location of the parameter called by the argument. Assignment of a value to the parameter effectively results in the value being placed in the parameter's storage location.

For example, the previous function call (above) will produce two different end results (for y) depending on whether the parameter x is called by reference or called by value.

	Call Method	End Result
1.	Call-by-reference	y=5
2.	Call-by-value	y=1Ø

In the first case, argument y is essentially substituted for parameter x. Every subsequent reference to parameter x actually becomes a reference to the storage location of argument y. Each assignment of a value to x is in effect an assignment of the same value to y. Thus y is passed through the function and returns a new value (5) to y's storage location. The argument y now has a new value in the external program.

Call-by-reference functions obviously change the value of an argument passed through them. This fact should be noted so that strange results emanating from a program containing user-defined functions do not unduly alarm the programmer.

In the call-by-value case, the argument y passes only its value (1Ø) to the dummy variable x. It does not itself pass through the function FNA. At the end of the pass, y remains unchanged because the parameter x does not return a value to argument y's storage location. In this example, y itself remains external to the function and thus retains its original value of 1Ø. It can be inferred that arguments cannot return values from a function pass in call-by-value systems.

Forcing Call-by-Value

It is possible to force an argument to set a parameter by value in a call-by-reference system, such as BASIC/VM. The parameter in the function definition is modified to create a temporary value inside the function which will be passed through with no affect on the calling argument's original value.

Example:

Forced Call-by-Value	Call-by-Reference
A = FNA(x+0)	A = FNA(x)
1) x is loaded into the accumulator	1) call FNA
2) 0 is added to it	2) argument pointer references x
3) the result is stored in a temporary storage location T1	3) the result is stored in A
4) call FNA	4) the arguments' storage location is updated. (direct correlation between argument and parameter x, no local value for x is created)
5) the argument pointer references T1 and the function operates on value in T1	
6) the value is then stored in A	
7) the arguments' storage location is not updated because the storage location for (x+0) is unchanged; the argument pointer references T1 only and is local to the function.	

The following program is an example of a call-by-reference function:

```
100 DEF FNA (X)
110 Y=X*X
120 X=Y+1
130 FNA = Y
140 FNEND
150 X=1
160 Y=2
170 Z=3
180 PRINT 'X': 'Y': 'Z': 'FNA(Z)', 'X': 'Y': 'Z'
190 PRINT X: Y: Z: FNA(Z), X: Y: Z
200 END
```

In this program, the following occurs:

1. The function of X is defined in lines 100-140. Since X is a dummy parameter, changing X in the function definition does not change the actual value of X.

2. Line 110 changes the variable Y to equal X2

3. Line 120 changes the argument which corresponds to the parameter X.

4. Line 130 sets the value of the function to Y.

5. Line 180 references the function by using the argument Z which is passed to parameter X. Therefore, when X changes, Z is also changing.

The resulting output is:

```
X Y Z FNA(Z)        X Y Z
1 2 3 9             1 9 10
```

Function Definitions and Program Control

The following program demonstrates the transfer of program control via function definitions and GOSUB statements:

```
  10  REM A PROGRAM WITH SUBROUTINES
  20  REM AND FUNCTIONS
  30  DEF FNA (X)
  40  IF X=0 THEN FNA=-1 ELSE FNA = X*X+J
  60  FNEND
  70
   .
   .
   .
 100  IF A=1 THEN 1000
   .
   .
   .
 150  X=FNA(3)
   .
   .
   .
1000  PRINT 'A EQUALS ONE'
   .
   .
   .
2000  GOSUB 5000
2010
   .
   .
```

```
   .
2500  Y=FNA(G)
3000  GOTO 100
   .
   .
   .
5000  Z=FNA(I)
5010
   .
   .
   .
6000  RETURN
```

Program executions begins at line 70. Assuming A=1, control jumps from line 100 to line 1000. At line 2000, control transfers to line 5000. Because the function is referenced, control transfers to lines 30 through 60 where the function is defined. Control then returns to line 5000 for the assignment and continues with 5010. After 6000, control returns to 2010. Functions may also be recursive (i.e., they may call themselves.) For example,

```
100 DEF FNF(X)     !Factorial
110 IF X<=1 THEN FNF = 1 ELSE FNF = X*FNF(X-1)
120 FNEND
```

Avoiding Function - I/O Interaction

When user-defined functions are included in a program, control statements should not be made from inside a function definition out nor from outside in. This can cause system stack difficulties leading to memory overflow. Additionally, when dealing with functions that perform I/O operations, (e.g., READs, WRITEs, PRINTs, INPUTs) the programmer should avoid calling these functions within other I/O statements. In other words, it is potentially confusing to the I/O handler to call a function to do a READ, for instance, while the function is being printed. The resulting printout will reflect this strange interaction of function READs and I/O PRINTs.

In the following example, the function FNI$ performs the I/O function READ in both programs. In the first program, WRONGPROG, the function is placed on the I/O list and is called to READ while it is being printed. The second program solves the problem of potentially ambiguous I/O by assigning the information returned by the function READ process to a temporary variable. Then this information will not be intermixed with the actual printout of the function.

Example:

```
WRONGPROG
10 ! THIS IS WRONGPROG
20 ! THIS PROGRAM MAY CONFUSE THE I/O HANDLER
30 !
```

```
55 ! THIS IS ONLY A PORTION OF A LARGER PROGRAM
60 DEF FNI$(A)  ! READS A STRING FROM FILE UNIT A
70 !
75 READLINE #A,X$
80 FNI$ = X$
90 !
100 !
110 FNEND
   .
   .
   .
180 ! READ FILE 'YYY', PRINT ON TERMINAL
200 A=1
210 DEFINE FILE #A = 'YYY'
220 FOR I = 1 UNTIL 1=2
230 PRINT FNI$(A)
235 ! NO TEMPORARY VARIABLE ASSIGNED TO FUNCTION READ
240 NEXT I
```

```
RIGHTPROG
10 ! THIS IS RIGHTPROG
20 ! THIS PROGRAM DOES NOT CONFUSE THE I/O HANDLER
30 !
55 ! THIS IS ONLY A PORTION OF A LARGER PROGRAM
70 !
75 READLINE #A,X$
80 FNI$ = X$
90 !
100 !
110 FNEND
   .
   .
   .
180 ! READ FILE 'YYY', PRINT ON TERMINAL
200 A=1
210 DEFINE FILE #A = 'YYY'
220 FOR I = 1 UNTIL 1=2
230 T$ = FNI$(A)  ! ASSIGN VALUE RETURNED BY READ FUNCTION
240               ! TO A TEMPORARY VARIABLE
250 PRINT T$
260 NEXT I
```

SECTION 11

NUMERIC DATA

INTRODUCTION

The properties of numeric data as supported by BASIC/VM are described in this section. Refer to Section 12 for string data properties.

BASIC/VM numeric data is double-precision and floating-point, having a level of accuracy to 13 places in the mantissa and two places in the exponent. Numeric items consist of constants, variables, arrays and functions. They are also known as operands because they are manipulated or operated on within a program.

A numeric constant is a numeric item whose value does not change (and cannot be changed) during program execution. A numeric variable is the representation of a value which may or may not change during program execution. Numeric arrays and matrices are one or two-dimensional arrangements of data in rows and columns and are detailed in Section 9. Numeric functions are discussed in Section 1Ø.

Numeric Constants

A numeric constant may be an integer, a decimal, or an exponent. A decimal may have an optional sign (+ or -), a decimal point or an exponent specifier. Exponents consist of the letter E (signifying base 1Ø), and an optional sign (+ or -) followed by one or two digits. Exponential representation in BASIC/VM is very flexible; for example, .ØØ1 can be written as 1E-3, .Ø1E-1, or 1ØØE-5.

Note

> If more than thirteen digits are generated during any computation, the result of that computation is automatically printed in E format. (If the exponent is negative, a minus sign is printed after the E: 1E-Ø4; if the exponent is positive, a plus sign is printed: 1E+Ø4.)

If decimal points are omitted, BASIC/VM assumes them to be located immediately to the right of the last significant (i.e., rightmost) digit. If the signs of either the constant or the exponent are omitted, BASIC/VM assumes them to be positive.

The following are examples of acceptable numeric constants:

12

-6.666

2.5E-2 (.Ø25)

Numeric Scalar Variables

A numeric scalar variable (also called simple numeric variable) is a single letter (A-Z), or a single letter followed by a single digit (Ø-9). Each variable represents a single numeric value; there are 286 possible numeric scalar variables. A numeric scalar variable is initialized automatically to Ø at the start of the BASIC/VM program that defines it. Examples of acceptable numeric scalar variables are:

A, A1, X.

Numeric Subscripted Variables : Arrays and Matrices

A numeric array or matrix is named by a simple numeric variable, e.g., A, A6. A simple variable followed by a parenthesized value or pair of values, is known as a numeric subscripted variable or an array element. A singly subscripted variable or array name represents an element in a one-dimensional array, e.g., A(5). A doubly subscripted variable indicates a two-dimensional array element e.g., A(3,5). In a two-dimensional array, the first subscripted value represents row location; the second value represents column. For example the array element in location A(3,5) is visualized as being in row 3, column 5, of array A.

In BASIC/VM, a matrix is that part of an array whose elements have non-zero subscripts. For example, an array dimensioned as A(5) actually has 6 elements; A(Ø)-A(5). Matrix A has only 5 locations: A(1)-A(5).

Matrices and arrays also are dimensioned with the DIM statement, e.g., DIM A(5). Matrices can be defined or redimensioned with the MAT statement. See Section 9 for details.

Distinguishing Variable and Array Names

A numeric variable and a numeric array may share the same name e.g., B, in a program. The use of parentheses distinguishes one from the other. For example:

B=2

defines and assigns a value to a numeric scalar variable, while

DIM B(2)

defines a numeric array, B, with a dimension of 2 rows by 1 column (assumed).

It is possible to have an array and a simple scalar variable with the same name in a program, but it is not possible to have both a one-and-two dimensional array with the same name, i.e. A(1) and A(1,1).

The following shows how numeric constants, variables, arrays, matrices and array or matrix elements are defined.

Type	Examples
Constant	12 12.5 5E2
Scalar Variable	A=3 A7=12
Array or Matrix	DIM A(6) DIM A(5,5)
Array or Matrix or Element	A(6)=5 A(3,5)=10

NUMERIC EXPRESSIONS

Numeric expressions are constructed from numeric operands including:

numeric variables or array (matrix) elements
numeric constants
references to a numeric function

and numeric operators, including:

arithmetic operators
relational operators
logical operators

Expressions can be evaluated by arithmetic or string operators to a single value which may be used elsewhere in the program; or, they can be evaluated logically or relationally to a value of true or false. Program control flow may be decided on the basis of this value, as in a conditional GOTO expression.

OPERATORS

A numeric operator may be one of three main types depending on the kind of operation it performs: arithmetic (unary or binary), relational, and logical.

Operators which require one operand are called unary. They indicate the sign, positive (+) or negative (-1), of a numeric item. Operators which require two operands are called binary. Table 11-1 lists all operators according to type. Operators are evaluated according to a set priority list. See Order of Expression Evaluation, following.

Table 11-1. Numeric Operators

	Operator	Meaning	Example
Arithmetic Operators:		******* *UNARY* *******	
	+	plus	+I
	-	minus	-I
		******** *BINARY* ********	
	+	addition	I+J
	-	subtraction	I-J
	*	multiplication	I*J
	/	division	I/J
	^ (or **)	exponentiation	I^2
	MOD	remainder from division (Modulus)	I MOD J
	MIN	lesser value	I MIN J
	MAX	greater value	I MAX J
Relational Operators:	=	equal	I=J
	<	less than	I<J
	>	greater than	I>J
	<= or =<	less than or equal to	I<=J
	>= or =>	greater than or equal to	I>=J
	<> or ><	not equal	I<>J
Logical Operators:	AND	logical "and"	I=J AND K$=L$
	OR	logical "or"	I=J OR I=K
	NOT	logical complement	NOT (I=J)

Arithmetic Operators

Arithmetic operators come in two flavors, unary and binary. Unary operators require only one operand. Therefore, if the operator is the minus sign (-), the value of the operand will be negative. If the operator is the plus sign (+), the value of the operand will be positive. The operand can be a parenthetical expression. Examples:

```
PRINT -(A+B)

PRINT -A

PRINT +B
```

Relational Operators

BASIC/VM has six relational operators:

Operator	Meaning	Examples
<	less than	X<Y
>	greater than	X1>Y1
=	equal	I=J1
<= or =<	less than or equal	J2<=J3
>= or =>	greater than or equal	Z>=10
<> or ><	not equal	D<>19

The interactions of relational operators and operands are referred to as relational operations.

Logical Operators

The three logical operators, AND, OR, NOT are used in forming logical expressions. Logical expressions can be composed of variables and/or relational operations connected by one or more logical operators. They are evaluated to true or false. Figure 11-1 illustrates the evaluation of logical expressions under different true-false conditions. An expression that is true has a value not equal to 0; if false, it has a value of 0.

Relational operations are used with logical operators to form logical expressions. The result of a logical expression evaluation (true or

false) can be used to determine the flow of control within a program. For example, the following simple program shows the use of relational operations and logical expressions to form conditions for control transfer.

```
10 INPUT A,B
20 IF A>B AND B<>0 GOTO 80
30 IF A<B GOTO 60
40 PRINT 'A=B'
50 GOTO 90
60 PRINT 'A<B'
70 GOTO 90
80 PRINT 'A>B'
90 END
```

The expression 'A>B' in line 20 is evaluated using the values input at line 10. If the expression is true, a GOTO is executed. If false, the next sequential statement is executed, and so on.

Order of Expression Evaluation

A numeric expression is evaluated in the order of operator priority. This is determined by rules of precedence in BASIC/VM. These rules of precedence are:

expressions in parentheses

system and user defined functions

^ (or **)

NOT, unary (+,-)

*,/,MOD

+,-

MIN,MAX

relationals (=,<,>,<=,>=,<>)

AND

OR

Parenthetical expressions are always evaluated first. Then, operators with higher precedence are evaluated before operators with lower precedence.

Example:

```
(A + B) /2
```

The addition, A + B, being within parentheses, is performed first, then the division by 2 is performed, even though the division operator has higher precedence. Operators with equal precedence are evaluated from left to right.

Example:

```
A + B - C * D * E ^ F ^ G
```

is interpreted as:

```
(A + B) - ((C * D) * (E^(F^G))
```

where the order of operation is:

1. A+B, C*D, F^G
2. The result of E^(F^G)
3. The result of C*D multiplied by the result of step 2.
4. The result of A+B minus the result of step 3.

AND

A	B	A AND B
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

OR

A	B	A OR B
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

NOT

A	NOT A
TRUE	FALSE
FALSE	TRUE

Figure 11-1 Logical Expressions

Evaluation of Logical Expressions

In a logical expression each expression is evaluated as true or false. The logical operators determine whether the entire expression is false or true:

Given the values:

E=Ø
A=6
C=3
B=2
D=7

E AND A-C/3	is evaluated as false since the first term in the expression is equal to false.
A+B AND A*B	is evaluated as true since both terms in the expression are true.
A=B OR C=SIN(D)	is evaluated as false since both expressions are false.
A OR E	is evaluated as true since one term of the expression (A) is true.
NOT E	is evaluated as true since E=Ø.

Note

Logical values can only be used in IF, WHILE, UNLESS, or UNTIL statements. Expressions such as LET A=B<>C or A=B AND C are not legal.

SECTION 12

STRING DATA

INTRODUCTION

A string is a sequence of ASCII characters. BASIC/VM string operands or data elements include constants, variables, arrays, matrices and functions. String array and matrix operations are covered in Section 9. String functions are detailed in Section 10.

String Constants

A string constant or string literal is a sequence of characters enclosed in single quotes or apostrophes ('). All spaces enclosed in the quotes are included in the string value. The value of a string constant does not change during program execution. The length of a string constant ranges from 0 characters (a null string, ''), to 160 characters.

Examples:

```
'12345.6'
'I am a string constant'
'' (a null string)
'Columbus, Ohio'
```

String Scalar Variables

A string scalar variable (or simple string variable) is a single letter (A-Z) followed either by a dollar sign ($) or by a decimal digit (1-9) and a dollar sign ($). String variables represent character strings of varying lengths and are initialized to the null value at the beginning of the program in which they are defined. Examples of string scalar variables are:

```
A$
X2$
```

The following is a program excerpt using string scalar variables to prompt for input:

Example:

```
10 !B$=USER NAME
20 A$='WHAT IS YOUR NAME'
30 PRINT A$;
40 INPUT B$
50 PRINT 'WELCOME TO BASIC,':B$
55 END
>RUNNH
WHAT IS YOUR NAME!BULLWINKLE
WELCOME TO BASIC, BULLWINKLE
```

String Subscripted Variables (Arrays): are simple string variables followed by one or two values enclosed in parentheses. Subscripted variables represent array or matrix elements. Singly subscripted variables, e.g., A$(1), indicate elements in a one-dimensional string array, e.g., A$. A doubly subscripted variable, e.g., A2$(1,2), represents an element in a two-dimensional array or matrix, e.g., A2$. In a doubly subscripted variable, the first value represents rows, the second, columns. For example, the array location represented by A$(1,2) is located in row 1, column 2.

String Arrays and Matrices

String arrays and matrices are dimensioned and defined by a DIM statement or a MAT statement. See Section 9 for details. For example:

```
DIM A$(6)
```

defines a one-dimensional array of seven elements: A$(0) through A$(6);

```
DIM A2$(3,4)
```

defines a two-dimensional array of four rows (0 through 3) and five columns (0 through 4).

All elements of a string array are character strings of variable length. For example:

```
10 DIM B$(2,2)
20 B$(1,1)='MICHELLE'
30 B$(1,2)='BECKY'
40 B$(2,1)='KAREN'
50 B$(2,2)='ARLENE'
55 MAT PRINT B$
60 END
>RUNNH
MICHELLE            BECKY
KAREN               ARLENE
```

Naming Variables and Arrays

String variables and arrays may have the same name in a program, e.g. B$. However, the way in which they are defined distinguishes a simple variable from an array. For example:

```
B$='HELLO'
```

defines and sets a value for the string scalar variable, B$;

```
DIM B$(2,3)
```

defines a two-dimensional string array of three rows and four columns.

The following are examples of string data types:

Type	Examples
Constant	'-125' 'CONSTANT VALUE'
Scalar Variable	B$
Array or Matrix	DIM A$(4,6)
Array or Matrix Element	A$(4,6) A$(5)

STRING EXPRESSIONS

String expressions are constructed of string operands including:

- string variables or array (matrix) elements
- string constants
- references to a string function

and string operators, including:

- concatenation operator
- relational operators
- logical operators

String Operators

The following are BASIC/VM string operators:

Operator	Meaning
+	concatenation (combines strings)
<	less than
>	greater than
=	equal
<= or =<	less than or equal
>= or =>	greater than or equal
<> or ><	not equal
AND	logical "and"
NOT	logical complement
OR	logical "or"

The above operators are the only legal string operators in BASIC/VM. String operators may not be used with numeric operands; similarly, numeric operators cannot be used on string operands.

The Concatenation Operator: is used to combine two or more string values to produce a single string.

Example:

```
1Ø A$='TODAY IS: '
2Ø PRINT 'ENTER TODAYS DATE: MONTH DAY YEAR'
3Ø INPUT LINE D$
4Ø PRINT B$=A$ + D$
45 END
>RUNNH
ENTER TODAYS DATE: MONTH DAY YEAR
!JANUARY 24, 1979
TODAY IS: JANUARY 24, 1979
```

Logical and Relational Operators: String data may be used in relational and logical expressions just as are numeric data. Logical expressions can be composed of both string and numeric relationals.

Evaluation of String Relational Expressions

Comparison of string expressions or values is conducted on a character by character basis. Ranking of characters is determined by ASCII code. See Appendix B for a complete list of characters and their decimal values. If the strings being compared are of different lengths, the shorter of the two is padded (internally) on the right with blanks until the strings are the same length. The strings are then compared, character by character, until the last common non-blank character position is reached in both strings. At this point, a decision is made on the basis of the relative decimal values of this last character.

For example, if the strings 'Z' and 'AZ' are compared, 'Z' is considered greater than 'AZ' because the decimal value of Z (value:218) is greater than the decimal value of A (value:193).

Example:

```
Ø5 PRINT 'THIS IS A COMPUTER TASTE-TEST'
1Ø A$='MICHELOB'
2Ø B$='MILLER'
3Ø IF A$>B$ GOTO 55
4Ø PRINT 'MILLER IS GREATER THAN MICHELOB'
5Ø GOTO 6Ø
55 PRINT 'MICHELOB IS GREATER THAN MILLER'
6Ø END
>RUNNH
THIS IS A COMPUTER TASTE-TEST
MILLER IS GREATER THAN MICHELOB
```

Lowercase letters have a higher decimal value than uppercase letters. For example:

```
1Ø A$='bad'
2Ø B$='BAD'
3Ø IF A$=B$ THEN PRINT 'EQUAL'
35 IF A$>B$ THEN PRINT 'A$ GREATER' ELSE PRINT 'A$ LESSER'
4Ø END
>RUNNH
A$ GREATER
```

If strings are to be compared on the basis of physical length and not relational value, use the LEN function:

Example:

```
10 A$='HI'
20 B$='HARVEY WALLBANGER'
30 IF LEN(A$) < LEN(B$) GOTO 60
40 PRINT 'WRONG'
50 PRINT
60 PRINT A$: 'IS SHORTER THAN':B$
65 END
>RUNNH
HI IS SHORTER THAN HARVEY WALLBANGER
```

Operator Priority

String expressions are evaluated according to operator priority. The rules of precedence are:

```
NOT
+
>,<,=,>=,<=,<>
AND
OR
```

Relational operators have equal priority and are evaluated in left to right order if more than one appears on a statement line. Parenthetical expressions are evaluated first. Within parentheses, evaluation proceeds according to operator priority. For example, in evaluating this expression:

```
IF A$<=B$+(C$(X)+X$) THEN GOTO 100
```

the following steps are taken:

1. The parenthetical expression (C$(X)+X$) is evaluated.
2. B$ is concatenated to the result of step 1.
3. A$ is compared to the result of step 2.
4. If A$ is less than or equal (on the basis of character rank) to the result of step 3, control transfers to line 100; If the condition is false, control transfers to the next sequential statement.

SECTION 13

PRIMOS SYSTEM COMMANDS

CONVENTIONS

The conventions for PRIMOS and BASIC/VM system commands are:

- WORDS-IN-UPPER-CASE

Capital letters identify command words or keywords. They are to be entered literally. If a portion of an upper-case word is underlined, the underlined letters indicate the minimum legal abbreviation.

- Words-in-lower-case

Lower case letters identify parameters. The user substitutes an appropriate numerical or text value. Hyphens connecting parameter phrases like the one above, are not literal components of the parameter.

- Braces { }

Braces indicate a choice of parameters and/or keywords. Unless the braces are enclosed by brackets, at least one choice must be selected.

- Brackets [ ]

Brackets indicate that the word or parameter enclosed is optional.

- Hyphen -

A hyphen preceding a parameter is a required part of that parameter or option, e.g, SPOOL -LIST.

- Parentheses ( )

When parentheses appear in a command format, they must be included literally.

- Ellipsis ...

The preceding parameter may be repeated.

- Angle brackets < >

Used literally to separate the elements of a pathname. For example: <FOREST>BEECH>BRANCH537>TWIG43>LEAF4.

- option

The word option indicates one or more keywords or parameters can be given, and that a list of options for the particular command follows.

- Underlines

For PRIMOS commands only, acceptable command abbreviations are underlined in the given formats.

ATTACH new-directory

new-directory is the pathname of the new working directory to which the user wants to be attached; becomes the current working directory. If any directories in the pathname are passworded, the entire pathname should be enclosed in single quotes, as in:

```
A 'FLOWER STEM>ROSE'
```

AVAIL [* | disk-number | packname]

Returns the number of normalized disk records available on a specified disk or the current disk (*), calculated at 440 words per record. The number of words per normalized record may not correspond to the number of words per physical record on the disk in question.

BASICV [pathname]

Invokes the BASIC/VM subsystem from PRIMOS command level. The system responds with the latest revision number and the query, NEW OR OLD:. Once a NEW filename or an OLD filename has been entered, the system responds with the BASIC/VM prompt character (>).

If the pathname option is given, this command runs the named BASIC/VM program and returns the user to PRIMOS command level.

CNAME oldname newname

Changes oldname, a filename or the last portion of a pathname identifying a sub-ufd or file, to newname, a new filename.

```
COMINPUT { pathname  }
         { -CONTINUE }
         { -END      }
         { -PAUSE    }
         { -START    }
         { -TTY      }
```

If pathname is specified, calls in and reads commands from the specified file (called a command file) rather than from the user's terminal; otherwise, one of the following control options is performed:

-CONTINUE
-START — Resumes execution of the command file after a -PAUSE.

-END
-TTY — Closes command file and causes PRIMOS to resume taking commands from the terminal. Either CO -END or CO -TTY should be the last command in the command file.

-PAUSE — Temporarily suspends execution of the command file. Allows commands to be given from the terminal without closing the command file.

```
COMOUTPUT { pathname  }
          { -CONTINUE }
          { -END      }
          { -NTTY     }
          { -PAUSE    }
          { -TTY      }
```

If pathname is specified, creates a file in which all terminal I/O is stored; otherwise, performs one of the following control options:

-CONTINUE — Continues command output to pathname.

-END — Stops command output to the specified file and closes command output file units.

-NTTY — Turns off terminal output. Does not display responses to command lines. Terminal output is resumed when COMO-TTY command is given.

-PAUSE — Stops command output to pathname; however, the command output file remains open.

-TTY Turns on terminal output. (default)

CREATE pathname

Creates a new file directory (sub-UFD) within specified directory. Two files with the same name are not allowed in the same directory.

DELETE filename

Deletes a specified file from the current UFD or sub-UFD. filename is any existing file or empty directory to be deleted. If a ufd-name, under which there are no files or sub-UFDs, is specified, the entire UFD will be deleted.

LISTF

Lists all entries under the current UFD, including all directories and files.

LOGIN ufd-name

Allows access to files and programs in a specified directory; ufd-name is the name of a login directory. The LOGIN command must be typed before any interaction with the system can take place. If no command is given and interaction is attempted, (or if the wrong ufd-name is given), PRIMOS responds with an error message. To a legal LOGIN command, PRIMOS responds with the terminal number, the current time, the current date, and finally the PRIMOS prompt 'OK,'.

LOGOUT

Terminates all interaction with PRIMOS.

PRIMOS responds to the command with the terminal number, the current time of day, and the amount of computer (CPU) time used.

PASSWD owner-password [nonowner-password]

Protects the current directory by specifying owner and nonowner (optional) passwords which are required in order to access (attach to) the directory.

PROTEC pathname [owner-rights, [nonowner-rights]]

Sets protection (access) rights on the file specified by pathname. owner-rights is an integer specifying owner's access rights to the file; nonowner-rights is an integer specifying nonowner's access rights to file. Access rights are listed below:

Access Rights

Ø No access of any kind
1 Read only
2 Write only
3 Read and write
4 Delete and truncate
5 Delete, truncate and read
6 Delete, truncate and write
7 All access

Default: Keys are 7 Ø (owner has all rights, nonowner has none).

```
SIZE pathname
```

Returns the size in records (decimal) of a file specified by pathname. The number of records per file is defined as the number of data words in the file divided by 440, rounded up.

```
SLIST pathname
```

Displays the contents of file specified by pathname at the terminal.

```
SPOOL { pathname       }
      { -CANCEL PRTxxx }
      { -LIST          }
```

If pathname is specified, causes the line printer to type out a specified file. PRIMOS assigns the file a number in the form PRTxxx, where xxx is a number between 001 and 200. The -LIST option returns a list of all users whose files are in the queue to be spooled. The list includes user name, filename, and file size. The -CANCEL PRTxxx option removes file identified by PRTxxx from the spool queue. Binary files cannot be spooled. Files are printed according to the time the file was spooled or according to file size.

```
STATUS { ALL     }
       { DISKS   }
       { NETWORK }
       { UNITS   }
       { USERS   }
```

Returns system status information indicated by specified options. ALL returns all information, including disk names and physical-to-logical disk correspondence (DISKS), network information (NETWORK), user information (USERS), and number of open file units on the current disk (UNITS).

TERM [option(s)]

The most commonly used options are:

-ERASE character	Sets user's choice of erase character in place of the default, ".
-KILL character	Sets user's choice of kill character in place of default, ?.
-XOFF	Enables X-OFF/X-ON feature which allows programs to halt without returning to PRIMOS command level. Programs can be halted by hitting CONTROL. Programs may be resumed at point of halt by hitting CONTROL-Q. Also sets terminal to full duplex (default value).
-NOXOFF	Disables X-OFF/X-ON feature (default).
-DISPLAY	Returns currently set values of erase and kill characters. Also displays current duplex setting, Break and X-ON/X-OFF status.

If no options are specified, the TERM command will return a complete list of TERM options. See Appendix D.

USERS

Returns the number of users logged into PRIMOS at any given time.

SECTION 14

BASIC/VM SYSTEM COMMANDS

The following is an alphabetized description of all BASIC/VM system commands. Commands are issued at BASICV command level, in response to the BASICV system prompt character, '>'. Each command must be typed in upper case and cannot be abbreviated. Some commands may also be used as statements and are appropriately indicated.

ALTER line-number

Enters an editing mode to allow modification of indicated line. Editing subcommands, listed below, are entered in response to the special ALTER mode colon (:) prompt. More than one such command can be packed into a single line; no delimiter is necessary. The colon prompt is returned until QUIT is typed.

Subcommand	Effect
A/string/	Append string to end of line.
Bnn	Move pointer back nn characters (where nn is any integer).
Cc	Copy line up to but not including c (where c is any character).
Dc	Delete line up to but not including c.
En	Erase n characters.
F	Copy to end of line.
I/string/	Insert string at current position. (the slash may be any delimiter not used as part of the string.
Mn	Move n characters.
N	Reverse meaning of next C or D parameter (copy until character =<c, or delete until character =>c).

O/string/	Overlay string on line from current position. A '!' changes a character to a space, a space leaves character unchanged.
Q	Exit from Alter mode.
R/string/	Retype line with string from current position. (Similar to Overlay but '!' and space have no special effects.)
S	Move pointer to start of line.

ATTACH pathname

Attaches to directory specified by pathname; may have one of the following formats:

Format 1: *>sub-ufd-name

where * indicates the current directory

Format 2: { <*> / <disk> } ufd-name[>sub-ufd-name]

where <disk> is the logical disk number on which directory named by ufd-name is located. <*> indicates current disk. More than one sub-ufd-name may be indicated if sub-ufds are nested.

Format 3: ufd-name [>sub-ufd-name]

where directory named by ufd-name is located on the current disk.

Example:

```
ATTACH<6>MANUALS>REV16>PRCGCOMP>BASICV
```

Although similar to the PRIMOS ATTACH command, this command may not be abbreviated and is issued at BASICV command level. If directories are passworded, the passwords must be included.

```
BREAK { ON  } lin-num-1 [,...lin-num-n]
      { OFF }
```

Sets and unsets breakpoints at specified statement lines for debugging. line-1 through line-n are statements at which the program is instructed to stop. A maximum of 1Ø may be set. The LBPS command returns a list of all currently set breakpoints.

If a statement line at which a breakpoint is set is reached during execution time, the program stops and returns to BASIC/VM command level; type CONTINUE to resume execution. BASIC/VM resumes execution with the statement specified by BREAK ON, and continues until the next breakpoint, STOP, END, or error is encountered.

If statement numbers are not specified with BREAK OFF, all previously set breakpoints are automatically turned off.

```
CATALOG [option(s)]
```

Lists all filenames under the current UFD, plus option information, if specified. option(s) are any or all of the following:

DATE	Returns the date and time of files' last modification.
PROTECTION	Returns the files' protection attributes (owner and nonowner rights). See Section 2.
SIZE	Returns the size of each file in records.
TYPE	Indicates whether the file is SAM, DAM, SEGSAM, SEGDAM, or a UFD.
ALL	Includes all of the above information.

If no options are specified, CATALOG returns only the filenames.

CLEAR

Resets all previously set numeric or string variables to zero or null respectively. Also deallocates previously defined arrays and closes all open files. Useful in Immediate mode calculations.

COMINP { pathname / CONTINUE / PAUSE / TTY }

Opens and reads commands in command file of specified pathname. If control options (CONTINUE, PAUSE) are specified, command file halts at COMINP PAUSE, resumes with COMINP CONTINUE. Commands in this file are executed until a COMINP TTY command is reached. This is generally the last command in the COMINP file. COMINP may also be used as a statement; see Section 15.

NOTE

As a command, COMINP takes an unquoted argument: as a statement, it takes a legal BASIC string argument.

COMPILE [pathname]

Translates the foreground source program into an executable binary program (machine language) which can be named and saved by specifying pathname. This binary file can be executed directly by specifying its pathname with EXECUTE. See EXECUTE. If the filename (pathname) is omitted, the system compiles the code into user memory for use with EXECUTE but no binary file is saved to disk. COMPILE also displays at the terminal any syntax errors (e.g., bad statement format, misspellings, etc) that may occur in the program. These are known as 'compile-time' errors, as distinguished from 'run-time' errors which occur during program execution.

```
CONTINUE
```

Resumes program execution after a PAUSE or a breakpoint.

```
DELETE {lin-num-1[,...lin-num-n]}
       {lin-num-1 - lin-num-n   }
```

Deletes the specified statement lines from program. lin-num-1 through lin-num-n are statement numbers to be deleted. Statements may be listed individually, separated by commas (as in first format), or they may be specified in a range (as in second format), the beginning and end of which are separated by a dash, as in 10-300.

```
EXECUTE [pathname]
```

If no pathname is specified, the currently compiled code in user memory is executed. If none exists, the foreground source file is translated into executable machine language and then executed. When a binary file pathname is given, the binary file is immediately executed. If a source file is specified, it is first compiled into machine language and then executed. EXECUTE also displays at the terminal any run-time errors that may occur during program execution. Run-time errors are usually logic or control errors which interrupt or inhibit program execution, e.g., a READ after a WRITE to a sequential file.

```
EXTRACT {lin-num-1[,...lin-num-n]}
        {lin-num-1  -  lin-num-n}
```

Deletes all except the specified lines. lin-num-1 through lin-num-n are statement numbers to be saved. Statements may be listed individually separated by commas, or they may be specified in a range, the beginning and end of which are separated by a dash. The statement numbers must be in ascending order.

```
FILE [pathname]
```

Saves all input and modifications to current file under original name (default), or under new name specified by pathname.

When filing a program, there are several points to remember:

1. If a pathname is not specified, BASIC/VM automatically uses the name of the foreground file.

2. If a pathname different from the original name is specified, you will have two versions of the same file. If the pathname already exists, BASIC/VM returns the prompt:

   OK TO REPLACE:

   All responses other than Y, YE, YES or OK, are interpreted as NO, and BASICV requests another pathname.

3. A program need not be complete to be FILEd. It is advisable to FILE from time to time to avoid losing file modifications due to an inadvertent typing error. However, be sure to FILE a modified program before calling in another file or exiting the system, lest it be overwritten, truncated or generally garbled.

```
LBPS
```

Lists currently set breakpoints. Breakpoints can be set by the BREAK ON command.

LENGTH

Reports the number of statements in the current program.

LIST[NH] [{lin-num-1[,...lin-num-n]} | {lin-num-1 - lin-num-n}]

Displays the contents of the foreground file at the terminal. NH option suppresses program header (date, title etc). lin-num-1 through lin-num-n are statement numbers which may be listed individually with command separators, or specified in a range, the beginning and end of which are separated by a dash.

LOAD pathname

Merges external file, specified by pathname, with foreground file. Line numbers in the external file which are duplicated in the foreground file are overwritten by those in the external file; otherwise, lines are inserted or appended in numerical sequence.

If the specified file is binary, it is loaded into user memory but does not become part of the foreground file. After a binary file LOAD, an EXECUTE with no pathname will run the just-LOADed binary file.

NEW [pathname]

Indicates to BASIC/VM that a new foreground file is to be created with the specified name. All lines previously in the foreground are erased.

OLD [pathname]

Calls an existing file, identified by pathname, to the foreground. The last component of the pathname is the name of the file being called to the foreground. In the following example, STARTREK is the file called to the foreground:

```
OLD GAMES>BASIC>JUNK>STARTREK
```

PURGE [pathname]

If pathname specified, deletes indicated file from directory. Default: deletes the disk copy of the foreground file. A file currently open cannot be PURGEd.

After the PURGE command is issued, the file remains in foreground until another file replaces it. PURGE can also be used as a statement; see Section 15.

QUIT

Returns control to PRIMOS from BASIC/VM command level. Unlike CTRL-P and BREAK, QUIT closes all files opened by BASIC/VM , and deletes temporary files created by BASIC/VM.

RENAME newname

Changes the name of the foreground file, but does not rename the original disk copy of the file. If the renamed file is FILEd, two copies of the file will exist with different names. The renamed file will not be saved unless it is FILEd.

RESEQUENCE [new-start] [,old-start] [,new-incr]

Renumbers statements in the foreground program. new-start is the number which begins the new sequence. (Default: 1ØØ). old-start is the existing line number at which to begin renumbering. (Default: lowest numbered line). new-incr specifies increment value. (Default: 1Ø).

RUN[NH] [lin-num]

Begins compilation and execution of the foreground source program, at lin-num, if specified. No binary file is stored by the RUN process. NH suppresses the program title, date and time usually displayed at run-time. RUN also displays all compile-time errors (statement syntax, spelling, etc) and run-time errors (faults in program logic) that may occur during program translation and/or execution.

TRACE { ON / OFF }

Displays in brackets all statement numbers as they are executed until the TRACE OFF command is typed. The statement numbers may be stored in a separate file (see the PRIMOS command COMOUT, Appendix D). TRACE ON is issued immediately after compilation (COMPILE) and immediately prior to program execution (EXECUTE). Used to examine program logic or control.

TYPE pathname

Displays the contents of the specified non-foreground file at the terminal, but does not replace the file currently in foreground.

SECTION 15

BASIC/VM STATEMENTS

The following is an alphabetized description of all BASIC/VM statements and their formats. Conventions are the same as for BASIC/VM system commands in Section 14. No abbreviations are accepted and all statements must be typed in upper case. Statements which can be used as commands are so indicated.

BASIC/VM Conventions

All PRIMOS command conventions, as listed in Section 2 or Section 13, apply also to BASIC/VM commands and statements. In addition, the following parameter representations are used throughout this section:

arg	function argument
con	constant (numeric or string)
(dim)	dimension for array or matrix; a numeric item
expr	an expression; i.e., a combination of operands and operators which can be evaluated. Can be either numeric (num) or string (str).
str	string
var	variable
unit	file unit number (a numeric constant) on which file is opened for reading and/or writing.

```
ADD #unit, str-expr-1, { PRIMKEY
                       { KEY zero-expr } = str-expr-2 keylist
                       { KEY
```

where keylist = [,KEY num-expr-1 = str-expr-3]*

Adds record, str-expr-1, to MIDAS file, opened on unit. A primary key, PRIMKEY, KEY zero-expr or KEY and its value, str-expr-2, must be supplied. One or more secondary keys may be specified in keylist, which contains the names, num-expr-1, and value(s), str-expr-3, of the secondary key(s). * indicates repetition of expression as necessary.

```
CHAIN pathname
```

Closes all open files and transfers program control to external program specified by pathname.

```
CHANGE num-array TO str-var
CHANGE str-expr  TO num-array
```

Transforms ASCII character string , str-expr, into a one-dimensional numeric array (num-array) containing the decimal values of the ASCII codes of the string, or transforms a numeric array of ASCII codes to its string equivalent, str-var. ASCII characters and their decimal equivalents are listed in Appendix B. For example, when A$= 'WORD' is changed to array A, A(Ø) contains the length of A$, or 4; A(1) contains the decimal code of W, i.e., 215, etc. Conversely, if array A is changed to A$, the resulting string length is controlled by the value in A(Ø).

```
CLOSE #unit-1[,...unit-n]
```

Closes file previously opened on unit by a DEFINE FILE statement. unit is maximum of 12.

```
CNAME oldname TO newname
```

Changes name of specified file. oldname is the pathname of the file to be renamed; newname is the new pathname or filename given to the file.

```
COMINP { pathname }
       { CONTINUE }
       { PAUSE    }
       { TTY      }
```

Stops execution of current program and executes commands from command file specified by pathname. COMINP PAUSE and COMINP CONTINUE temporarily halt and restart the command file respectively. Commands in file are executed until COMINP TTY, the last command in the file, is reached. Also used as a command; see Section 14.

```
DATA item-1[,...item-n]
```

Lists numeric and string constants to be accesed by a READ statement. For example, given the following READ and DATA statements:

```
DATA 12, 45, 'BULL'
READ A, B, C$
```

The variables A, B and C$ will be assigned the values 12, 45 and BULL, respectively. There may be any number of DATA statements within the same program.

```
DEFINE [READ  ] FILE #unit = filename [,type-code] [,record-size]
       [APPEND]
```

Opens file, named by filename, a string expression, on specified unit. Optionally assigns file type and access method, indicated by type-code. Type-codes are listed in Table 15-1, following. If no type-code is given, the default (ASC) is assumed. The default record length of 60 words (120 characters) may be increased or decreased by specifying record-size, (a numeric expression) in number of words. For MIDAS files, record-size should be set equal to the combined length of the data record and the primary key; this is specified during CREATK when the template is being created. Access may be restricted to read or append only with the READ and APPEND options respectively. A file DEFINEd as a READ file is assumed to exist.

```
DEFINE SCRATCH FILE #unit [,file-type] [,record-size]
```

Opens a temporary file on specified unit, of any type except MIDAS. When unit is closed, the scratch file is automatically deleted.

```
DEF FN var [(arg-1,...arg-n)] = expression
```

Defines a one line function named by var, a string or numeric variable, (No FNEND statement necessary.) Arguments (arg-1 to arg-n) are numeric or string scalar variables only.

```
DEF FN   var [(arg-1,...arg-n)]
   .
   .
FNEND
```

Defines a user-defined numeric or string function, of one or more lines. The last line must be FNEND. var is a simple numeric or string variable. arg-1 to arg-n are dummy arguments for the function; may be numeric or string scalar variables. The defined function is not executed until it is referenced in the program; control then shifts to the function definition until FNEND is reached.

Table 15-1. File Type-Codes

Type-Code	Access Method	Contents
ASC (default)	SAM	ASCII data, formatted like terminal output, using BASICV PRINT conventions, e.g., commas, colons and semi-colons, all dictate the appropriate number of spaces to be used as data delimiters. Records variable-length and easily inspected.
ASCSEP	SAM	ASCII data stored with commas inserted as data delimiters. Data are stored and read back exactly as entered. Records fixed-length, accessed sequentially.
ASCLN	SAM	ASCII data with comma delimiters, and line numbers inserted in increments of 1Ø at the start of each record. Designed to be edited at BASICV command level.
ASCDA	DAM	Similar to ASCSEP. Records fixed-length and blank-padded as necessary. Direct access method used for quick, random access to any record in the file.
BIN	SAM	Data storage transparent to user. Records are fixed-length, accessed sequentially. String data stored in ASCII code: numeric data stored in four-word floating-point form. Provide maximum precision and compactness of numeric data, but cannot be inspected by TYPE etc.
BINDA	DAM	Same as BIN but direct access method is used for random record access. Records not data-filled are zeroed out.
SEGDIR	SPECIAL	Identifies file as a segment directory. Subordinate files, identified by number, may be SAM, DAM or other SEGDIR files. An additional DEFINE is required to access a subordinate file.
MIDAS	SPECIAL	Multiple Index Data Access files. Created by Prime-supplied MIDAS utilities.

```
DIM var{(num-con)                }
       {(num-con-1, num-con-2)   }
```

Defines the dimensions of a numeric or string array or matrix, named by a numeric or string variable, var. Dimensions are represented by (num-con) and (num-con-1, num-con-2), numeric constants. Default: (10) or (10,10). Variables are not legal dimension specifiers in DIM statements. The lowest element of an array is always (0) or (0,0). Arrays and matrices may be redimensioned within a program with the MAT statement.

```
DO
 .
 .
DOEND
[ELSE DO]
 .
 .
[DOEND  ]
```

Sets up a series of statements in association with IF-THEN statements, executed if a specified condition is met. DOEND indicates the end of the series. ELSE DO is an optional alternative to previous set of DO statements. ELSE can also be used in conjunction with IF. (See IF statement). Dots (.) represent statements in program.

```
END
```

Terminates program execution: serves as messageless STOP.

```
ENTER time-limit, time-var, var
```

Allows a specified number of seconds, time-limit, (range 1 to 18ØØ), for user input of a value for a numeric or string variable, var. No prompt is given. time-var, a numeric variable, returns the actual time taken to enter value. Only one value can be entered from the terminal.

```
ENTER # user-num-var [,time-limit, time-var, var]
```

Returns user number assigned at LOGIN in a numeric variable, user-num-var. Other options are same as for ENTER.

```
ERROR OFF
```

Turns off all error traps in conjunction with the ON ERROR GOTO mechanism.

```
FOR index = start TO end [STEP incr]
```

Specifies beginning of loop. Used with NEXT statement. The loop index, which changes during program execution, is specified by index, a numeric variable. The initial value of the index is set to start, a numeric expression; the increment value is set by incr: and the final value of the index is represented by end, a numeric expression. When the index attains this value, loop execution stops.

```
FOR index = start [STEP incr] { WHILE } condition-expr
                              { UNTIL }
```

Specifies the beginning of a loop with statement modifier. Used in conjunction with NEXT. condition-expr, a conditional expression, determines how long the loop will be executed. The WHILE modifier indicates that loop execution will continue as long as the specified condition remains true. UNTIL specifies that loop execution will continue until the specified condition is met. start represents the initial index value; incr optionally sets the increment value. The default STEP size is zero. The following are examples of legal and illegal loop nesting. Example of legal nesting techniques are shown first:

Two-level Nesting

```
FOR I1   UNTIL I1=13
  FOR I2 = 1 TO 13
     .
     .
     .
  NEXT I2
NEXT I1
```

Three-level Nesting

```
FOR I1 = 1 TO 1Ø
  FOR I2 = 1 TO 1Ø
     FOR I3 = 1 TO 1Ø
        .
        .
        .
     NEXT I3
  NEXT I2
NEXT I1
```

Examples of unacceptable nesting techniques are:

Two-level Nesting

```
FOR I1 UNTIL I1=13
  FOR I2 = 1 TO 10
      .
      .
      .
NEXT I1
  NEXT I2
```

Three-level Nesting

```
FOR I1 = 1 TO 10
  FOR I2 = 1 TO 10
     FOR I3 = 1 TO 10
         .
         .
         .
NEXT I1
  NEXT I2
     NEXT I3
```

Note

The statement modifiers WHILE, UNTIL and UNLESS may be used with other executable statements as well. Note that UNLESS may not be used with FOR loops.

GOSUB lin-num

Unconditionally transfers program control to an internal subroutine beginning at specified lin-num. A RETURN must be executed to terminate the subroutine. Up to 16 GOSUB statements may be nested.

GOTO lin-num

Transfers program control forward or backward to a specified lin-num. A loop may be created when the specified line number appears prior to the GOTO statement. May be used with IF.

```
IF  expr { GOTO lin-num-1      }
         { THEN lin-num-1      }  [ELSE { statement-2 }]
         { THEN statement-1    }        { lin-num-2   }
```

Transfers program control depending on the value of a relational, logical or numeric expression (expr). lin-num is the statement number to which program control is transferred if the expression is true. statement-1 is executed if the preceding expression is true. If the expression is not true, either statement-2 will be executed, or control will transfer to lin-num-2, depending on which, if any, is specified. If expr is not true, and no alternative is provided, the next sequential statement is executed.

IF statements may be nested to any level. IF may be used in one of the following combinations:

1.
```
IF expr { GOTO line      }
        { THEN line      }  [ELSE { line      }]
        { THEN statement }        { statement }
```

2.
```
IF condition THEN DO
   .
   .
   .
DOEND
[ELSE DO
   .
   .
   .
DOEND]
```

```
INPUT ['prompt-string',] var-1[,...var-n ]
```

Prompts user for input specified by var-1 through var-n which are either numeric or string variables or array elements, separated by commas. If no prompt string is provided, the default prompt character (!) is given; otherwise, the string is printed.

Trailing commas are ignored as are data in excess of variables specified. Data must be input in the same order in which the variables are given and must also match the variable type, or an INPUT DATA ERROR will occur.

```
INPUT LINE ['prompt-string',] str-var
```

Prompts user, with optional 'prompt-string', for str-var, a string variable or string array element. Accepts entire input line, including colons, commas, and leading blanks as one entry.

```
[LET] var = expr
```

The assignment statement, used to assign values to numeric or string variables or array elements; the keyword LET is optional. var represents a numeric or string variable or array element. expr is either a numeric value, string expression or another variable.

```
MARGIN {value}
       {OFF  }
```

Sets number of characters per line to value, a numeric expression. Range is 1 to 32767; the default is 80. MARGIN OFF turns off all margin settings other than the default.

MAT mat = { ZER / CON / IDN / NULL } [(dim-1) / (dim-1,dim-2)]

Sets initial value of matrix elements to zero, one, identity or null, respectively. Also used to redimension a one-dimensional matrix to (dim-1), a numeric expression, or a two-dimensional matrix to (dim-1,dim-2). NULL can only be used on string matrices; it initializes all elements to a null value. IDN transforms a matrix into an identity matrix, one in which all elements, except those on the main diagonal, are Ø; the main diagonal elements each have a value of one (1). ZER initializes all matrix elements to zero. CON sets all elements equal to 1.

MAT mat-3 = mat-1 { + / - / * } mat-2

Adds, subtracts or multiplies the elements of mat-1 and mat-2 to form a target matrix, mat-3. In multiplication, the target matrix dimensions are the number of rows of mat-1 and the number of columns of mat-2.

Rules:

1. For addition and subtraction, the two matrices must have the same dimensions, e.g., DIM A(2,2), DIM B(2,2).

2. For multiplication, the number of columns in the first matrix must equal the number of rows in the second matrix. The result will be a matrix with the dimensions of the number of rows of the first matrix by the number of columns of the second matrix.

3. A matrix may not be multiplied by itself; nor can the current value of the target matrix (the one appearing on left side of equation) be used in the multiplication expression. For example, MAT A= A*C is illegal.

```
MAT mat-1 = (expr) * mat-2
```

Multiplies each element of mat-2 by a specified numeric value (expr) and assigns results to mat-1. If mat-1 is an existing matrix, its elements will be redefined, and its dimensions will be changed to those of mat-2.

```
MAT mat-1 = INV(mat-2)
```

Assigns the inverse values of a square matrix mat-2, (determinant not equal to Ø) to the target matrix, mat-1. The resulting values in mat-1 can be multiplied by mat-2 to yield the identity matrix in which all elements are equal to 1.

```
MAT mat-1 = TRN (mat-2)
```

Calculates the transpose of the values of mat-2 and assigns them to target matrix mat-1. A matrix is transposed by rotating it along the main diagonal. For example:

```
1 4 7        1 2 3
2 5 8        4 5 6
3 6 9        7 8 9

mat-2        mat-1=TRN(mat-2)
```

```
MAT INPUT ['prompt-string',] mat-1 [,mat-2][,...{mat(*)}]
                                               {mat-n  }
```

Reads data from the terminal and assigns the values to specified matrices, mat-1 through mat-n. mat (*) indicates that elements may be input until a new line is typed. Matrix is automatically dimensioned to number of input elements. Default prompt character is !, unless prompt-string is specified. The type of data input must match the matrix type (i.e., numeric or string).

MAT PRINT mat-1 [,...mat-n]

Prints indicated matrices, mat-1 to mat-n, at terminal. If a matrix name is followed by a colon instead of a comma, the elements will be separated by spaces instead of column tabs when printed. If more than one matrix is listed, each begins on a new line. Commas are the only delimiters which put matrix elements into columns, i.e., all columns, or one print zone, apart. If .NL. (new line) is typed after each input, output will occur in row order.

MAT READ mat-1 [,...mat-n]

Reads values from a data list and assigns them to the elements of the specified matrix or matrices. Values are assigned until all matrices are filled, or the data list is exhausted.

MAT READ [*] #unit, mat-1 [,...mat-n]

Reads data items from an external file opened on unit and assigns them to elements of specified matrix or matrices. Optional * indicates that all data from current record should be read before a new record is read.

MAT WRITE #unit, mat-1 [,...mat-n]

Writes an entire matrix or matrices to a file on the specified unit. If matrix names are followed by colons instead of commas, elements of the matrices are output one space apart instead of 21 spaces (one print zone) apart.

If two units have been opened, a matrix may be read from one unit and written to the other. For example:

```
MAT READ #1, A
MAT WRITE #2, A
```

NEXT num-var

Defines the end of a loop beginning with a FOR statement. The num-var matches the variable used with the companion FOR statement.

ON num-expr GOSUB lin-num-1,...lin-num-n

Transfers program control to a subroutine at a specified line number depending upon the value of the numeric expression, num-expr. When a RETURN statement is reached in the subroutine, control returns to the statement following the ON...GOSUB statement.

The value of the numeric expression must be less than or equal to the number of statement lines listed. Thus, if the value of the expression is 1, control will be transferred to the statement indicated by lin-num-1. If the value is n, control will be transferred to the statement indicated by lin-num-n. If the value of the expression is out of range, an error message will be displayed.

ON num-expr GOTO lin-num-1,...lin-num-n

Transfers program control to one of a list of line numbers (lin-num-1 to lin-num-n) depending on the value of the numeric expression, num-expr. If the value of num-expr is 1, control transfers to the first line number given, lin-num-1; if the value is 2, control transfers to the second line number given, and so forth. The value of num-expr must be less than or equal to the number of statement lines listed. If the expression value exceeds the number of lines listed, an error message is displayed.

ON END #unit GOTO lin-num

Establishes a line number to which program control will transfer when an END OF FILE occurs on specified unit. This statement does not test for END OF FILE; instead, it establishes the action to be taken when the end of the last record in a file is reached during a READ, POSITION, or other I/O operation.

ON ERROR GOTO lin-num

Establishes a line number to which program control will transfer when a run-time error occurs. Two variables, ERR and ERL, and the function ERR$(num-expr), are associated with ON ERROR GOTO.

ERR	Variable set to the code number of the error which activated the ON ERROR statement.
ERL	Line number being executed when the error occurred.
ERR$(num-expr)	Outputs actual text of the error message associated with an error code represented by a numeric expression, num-expr.

The ERROR OFF statement can be used to cancel all error traps set by ON ERROR GOTO statements. See ERROR OFF.

ON ERROR #unit GOTO lin-num

Establishes a statement line to which program control will transfer when an I/O error occurs on the specified unit, e.g., when an invalid number is entered.

```
PAUSE
```

Acts as an executable BREAK command. Suspends program process at line where PAUSE occurs. To resume program, type CONTINUE.

```
POSITION #unit TO record-number
```

In direct access files (ASCDA, BINDA), positions the internal record pointer to a specified record-number in a file on the specified unit. When pointer is positioned past last record, the ON END #unit GOTO statement is activated (if specified) or the error message, END OF FILE, is displayed.

```
POSITION #unit, KEY { [num-expr] = str-expr }
                    { SEQ                   }
                    { SAME KEY              }
```

Positions a file read pointer to a specified record in a MIDAS file. If a secondary key number, num-expr is not indicated, num-expr = Ø is assumed. If SEQ is supplied in lieu of key, the pointer positions to the next sequential record SAME KEY positions to datum only if next key matches current one. POSITION is similar to READ except that no data is retrieved.

```
PRINT [item-1,[{LIN}(num)],...item-n,[{LIN}(num)]]
              {TAB}                   {TAB}
              {SPA}                   {SPA}
```

Prints formatted information at the terminal. Item-1 to item-n represent numeric and/or string values.

LIN forces the specified number (num) of carriage return - line feed combinations between items in the output if num is greater than Ø. If num is less than Ø, it forces that many line feeds only: if num = Ø, only a (CR) is generated.

TAB forces tab to specified column number. SPA forces number (num) of spaces between items in output.

A comma in a print list causes the terminal to advance to the first character position of the next print zone. Each print zone consists of 21 character positions. If data will not fit on one line, it is continued on the next line. If a colon is used instead of a comma, the items are separated by a single space in the output; if a semicolon is used, no spaces are inserted between items.

When a numeric expression is printed, if the value of the expression is positive, the sign is suppressed. If the value of the expression is negative, a minus sign is printed for the sign character.

If used without any parameters, the PRINT statement causes a blank line in the output.

PRINT USING format-string, item-1[,...item-n]

Generates formatted output according to format characters in format-string, including a dollar sign, plus or minus signs, decimal points and right-left justification. Item-1 through item-n represent string or numeric values. Format characters listed in Table 15-2.

Table 15-2. Numeric Format Field Characters

Sample Item:	Using this format Specification:	Will be printed as:	Remarks:
POUND SIGN FORMAT SPECIFICATIONS (#)			
25	#####	25	Digits right justified in field with leading blanks
-3Ø	#####	3Ø	Sign is ignored because item is positive.
1.95	#####	2	Only integers are printed; the number is rounded off.
598745	#####	*****	If number is too large for the specified field, asterisks are printed.
PERIOD (DECIMAL POINT) FORMAT SPECIFICATIONS (.)			
2Ø	#####.##	2Ø.ØØ	Positions to right of decimal point are zero filled.
29.347	#####.##	29.35	Item is rounded off.
789Ø12.344	#####.##	*****	If number is too large for the specified field asterisks are printed.
COMMA FORMAT SPECIFICATIONS (,)			
3Ø.6	+$,###.##	+$ 3Ø.6Ø	A space is substituted for comma when the leading digits are blank.
2ØØØ	#,###.	2,ØØØ.	Comma is printed in indicated position.

ØØØ33	++##,###	+ØØ,Ø33	Comma is printed when the leading zeros are not suppressed.

VERTICAL (UP ARROW) FORMAT SPECIFICATIONS (^)

17Ø.35	+##.##^^^^	+17.Ø3E+Ø1
1.2	+##.##^^^^	+12.ØØE-Ø1
6ØØ2.35	+###.##^^^^	+6ØØ.23E+Ø1

Note

If more than four up arrows are used, the corresponding number of exponent digits will be printed.

PLUS SIGN FORMAT SPECIFICATIONS (+)

2Ø.5	+##.##	+2Ø.5Ø	Plus sign printed where indicated; item is positive.
1.Ø1	+##.##	+ 1.Ø1	Leading zero's print as blanks.
-1.236	+##.##	- 1.24	Minus sign printed when when item is negative.
-234.Ø	+##.##	*****	If number is too large for the specified field, asterisks are printed.

MINUS SIGN FORMAT SPECIFICATIONS (-)

2Ø.5	###.##-	2Ø.5Ø	Sign discarded if item positive.
ØØØ.Ø1	###.##-	.Ø1	Leading zeros immediately to the left of the decimal point are suppressed. Minus sign not printed when item is positive.

-234.Ø	###.##-	234.ØØ-	Sign printed where indicated.
-2Ø	---.##	-2Ø.ØØ	Floating minus signs are treated as digit positions.
-2ØØ	---.##	******	Number does not agree with the format; asterisks are printed.
2	---.##	2.ØØ	Item is positive; minus sign suppressed.

DOLLAR SIGN FORMAT SPECIFICATIONS ($)

3Ø.512	$###.##	$ 3Ø.51	Dollar sign printed
-3Ø.512	$###.##+	$ 3Ø.51-	Negative item; minus sign printed where indicated.
13.2Ø	+$$$$#.##	+ $13.2Ø	Floating dollar sign printed immediately prior to leftmost significant digit.

Table 15-3. STRING FORMAT FIELD CHARACTERS

POUND SIGN (#) AND ANGLE BRACKETS (<,>) FORMAT SPECIFICATIONS

Sample item:	Using this format specification:	Will be printed as:	Remarks
TWELVE	>######	TWELVE	Right-justified
TWELVE	<######	TWELVE	left-justified
GRAND	####	GRAN	Only 4 characters will fit into specified field. field.

READ var-1[,...var-n]

Reads numeric or string values from one or more DATA statements within the program, beginning with the lowest numbered one. var-1 through var-n are string or numeric variables separated by commas. Begins accepting values with first item in lowest numbered DATA statement. READ is always associated with one or more DATA statements. If the data items are exhausted before all variables are satisfied, an error message is displayed. The RESTORE statement may be used to recycle data values within a program.

READ [KEY] #unit [,KEY[num-expr]=str-expr | SEQ | SAME KEY], str-var

Reads data from specified record in MIDAS file on unit. Data is read into str-var. If READ KEY is specified, the key value is read into str-var. Num-expr and str-expr are the key numbers and values, respectively, of the primary or secondary key. SEQ reads next sequential record. SAME KEY returns datum only if next key matches current one.

READ LINE #unit, str-var

Accepts entire line of text (including commas and colons) as one data item and puts it in str-var. Reads from a record in a file previously opened on unit. When the statement has been executed, the internal record pointer automatically moves to the next record.

```
READ #unit, var-1[,...var-n]
```

Forces program to read a new record from the file previously opened on unit. var-1 through var-n are values to be read from current record. READ accepts value of the first variable in the record to which pointer is positioned. Pointer automatically moves to the next record after indicated values have been read.

```
READ * #unit, var-1[,...var-n]
```

* signals program to continue reading data in current record before new one is read. var-1 through var-n are values to be read from current record and subsequent records, as necessary to satisfy variables listed.

```
REM string
```

Indicates remark to reader; ignored by system. Exclamation point (!) is substituted for REM when comments are added to executable statements.

```
REMOVE #unit [, KEY[num-expr] = str-expr] +
```

Deletes specified key from MIDAS file. If primary key, num-expr = Ø, is specified, data associated with key are removed also. Multiple keys may be deleted with one statement line; + indicates key specification may be repeated one or more times.

```
REPLACE #unit SEG x BY SEG y
```

Deletes file referenced by indicated segment (SEG x) on segment directory opened on indicated unit. Pointer at SEG y (segment y) is moved to segment x; old pointer at SEG y is zeroed.

```
RESTORE [ # ]
        [ $ ]
```

Instructs program to reuse list of data items beginning with first item in lowest numbered DATA statement. Numeric data items are reused by specifying #; string items, by $. Both numeric and string items are reused if neither symbol is specified. RESTORE must precede READ statement indicating data items to be reused.

```
RETURN
```

Causes control to be returned from GOSUB subroutine. For every GOSUB in a program, exactly one RETURN must be executed.

```
REWIND #unit-1 [,unit-2,...unit-n]
```

Repositions record pointer to top of file on specified unit or units.

```
REWIND #unit [,KEY num-expr]
```

Rewinds pointer to beginning of MIDAS file opened on unit, at column specified by KEY num-expr. If num-expr=Ø or is unspecified, pointer is positioned to primary key (default).

```
STOP
```

Causes termination of program execution. Returns message: STOP AT LINE lin-num.

```
UPDATE #unit, str-expr
```

Writes string expression, str-expr, to current MIDAS file open on unit. Beware of changing keys with UPDATE if keys are being stored in record. BASICV does not monitor record composition and is not aware of changes made to key fields within a record. UPDATE is not equivalent to a REMOVE followed by an ADD.

```
WRITE #unit, item-1[,...item-n]
```

Writes data, string or numeric, specified by item-1 through item-n, (string or numeric variables), into the current record or output device opened on unit. If no values are specified, a blank line appears in the output. If a sequential file is closed after WRITE statement, all subsequent records in file are truncated.

```
WRITE #unit USING format-string, item-1[,...item-n]
                        OR
WRITE USING format-string, #unit, item-1[,...item-n]
```

Generates formatted output, determined by format characters in format-string, including tabs, spaces, and column headings. Output is written to current record or output device opened on unit. item-1 through item-n are numeric or string variables or expressions. See Tables 15-2, 15-3 for format characters. If items are separated by colons instead of commas, they are printed one space apart rather than tabbed to the next print zone. Semicolons as separators cause items to be printed with no intervening characters or spaces.

APPENDIX A

SAMPLE PROGRAMS

SAMPLE PROGRAMS

BASIC/VM's flexible control structure and unique string handling capabilities make it easily adaptable to many applications. The three sample programs presented in this appendix utilize most of the features discussed earlier in the manual. The first program enables you to plot and print out a graph. The second can be used to test math skills, and the third performs simple text formatting.

Sample Pogram 1:

GRAPHICS PROGRAM

```
100 !    GRAPH-DRAWING PROGRAM
110 !
120 ! GRAPH PROGRAM
130 !
140 ! SET UP ARRAYS
150 DIM C(2) ! C(1) = # OF HORIZ CHAR, C(2) = # VERT CHAR
160 DIM M(2) ! M(1) = X MIN, M(2) = X MAX
170 DIM N(2) ! N(1) = Y MIN, N(2) = Y MAX
180 DIM P(120,120) ! POINT ARRAY, P(I,J) = 1 IF POINT IS DEFINED
190 DIM X(100) ! X VALUES
200 DIM Y(100) ! Y VALUES
210 !
220 DEF FNB(P$)
230     FNB  = VAL(LEFT(P$,INDEX(P$,' ')-1))
240     P$   = RIGHT(P$,INDEX(P$,' ') + 1)
250 FNEND
260 !
270 PRINT 'TYPE INPUT FILE NAME.'
280 INPUT F$
290 DEFINE READ FILE #1 = F$, ASC
300 !
310 ON END #1 GOTO 400
320 Z = 0
330 FOR I = 1 STEP 1 WHILE Z = 0
340     READ #1, X$
350     X$ = CVT$$(X$,24) + ' '
360     X(I) = FNB(X$)
370     Y(I) = FNB(X$)
380 NEXT I
390 !
400 N = I - 1
410 PRINT 'DO AUTO SCALING?'
420 INPUT A$
430 !
440 PRINT 'TYPE (# OF HORIZONTAL CHAR., # OF VERTICAL CHAR.)'
450 INPUT C(1), C(2)
```

```
460 IF A$ = 'YES' THEN DO
470    M(1) = X(1)
480    M(2) = X(1)
490    N(1) = Y(1)
500    N(2) = Y(1)
510    FOR I = 2 TO N
520       IF X(I) > M(2) THEN M(2) = X(I)
530       IF X(I) < M(1) THEN M(1) = X(I)
540       IF Y(I) > N(2) THEN N(2) = Y(I)
550       IF Y(I) < N(1) THEN N(1) = Y(I)
560    NEXT I
570    DOEND
580 ELSE DO    ! MANUAL SCALING
590    PRINT 'TYPE MIN X, MAX X'
600    INPUT M(1),M(2)
610    PRINT 'TYPE MIN Y, MAX Y'
620    INPUT N(1), N(2)
630    DOEND
640 !
650 ! SET SCALE FACTORS
660 K = (C(1) - 1)/(M(2) - M(1)) ! X SCALE FACTOR
670 L = (C(2) - 1)/(N(2) - N(1)) ! Y SCALE FACTOR
680 A = (M(2) - M(1)*C(1))/(M(2) - M(1))
690 B = (N(2) - N(1)*C(2))/(N(2) - N(1))
700 MAT P = ZER    ! CLEAR POINT ARRAY
710 !
720 FOR I = 1 TO N   ! FILL POINT ARRAY
730     R = INT(K*X(I) + A + .5)
740     S = INT(L*Y(I) + B + .5)
750     IF R>0 AND R<=C(1) AND S>0 AND S<=C(2) THEN P(R,S) = 1
760 NEXT I
770 !
780 ! PRINT THE GRAPH
790 !
800 FOR J = C(2) TO 1 STEP -1
810     X$ = ''  ! BLANK OUT THE LINE BUFFER
820    FOR I = 1 TO C(1)
830        IF P(I,J) = 1 THEN X$ = X$ + '*'
840        IF P(I,J) = 0 THEN X$ = X$ + ' '
850    NEXT I
860    PRINT 'I':X$
870 NEXT J
880 X$ = ''
890 FOR I = 1 TO C(1) + 2
900     X$ = X$ + '-'
910 NEXT I
920 PRINT X$
930 END
>TYPE XXX
3 4
1 2
2 2
3 3
```

```
5 5
8 5

>RUN
GRAPH      FRI, JAN 05 1979          15:55:54


TYPE INPUT FILE NAME.
!XXX
DO AUTO SCALING?
!YES
TYPE (# OF HORIZONTAL CHAR., # OF VERTICAL CHAR.)
!30,10
I                  *           *
I
I
I          *
I
I
I          *
I
I
I *    *
-------------------------------
```

Sample Program 2:

MATH DRILL PROGRAM

```
100 !   MATH DRILL PROGRAM
101 !
110 DIM S$(3)
120 S$(1) = '+'  ! INITIALIZE SYMBOL ARRAY
130 S$(2) = '-'
140 S$(3) = 'X'
141 !
150 ! DEFINE FUNCTION TO GENERATE RANDOM OPERANDS.
160 DEF FNA(I,J) = INT(I*RND(0) + J)
161 !
170 R = 0  ! R -> # ANSWERS CORRECT
180 PRINT 'HELLO, WHO ARE YOU?'
190 INPUT N$
200 PRINT 'OK, ':N$:' I HAVE SOME MATH PROBLEMS FOR YOU.'
210 PRINT 'WHICH TYPE OF PROBLEMS WOULD YOU LIKE?'
220 PRINT '1. ADDITION'
230 PRINT '2. SUBTRACTION'
240 PRINT '3. MULTIPLICATION'
250 PRINT '4. MIXED'
260 PRINT
270 PRINT 'TYPE 1, 2, 3, OR 4'
280 INPUT T  ! T IS PROBLEM CLASS
290 PRINT 'HOW MANY SECONDS SHALL I GIVE YOU TO ANSWER EACH PROBLEM?'
300 INPUT U
310 S = VAL(SUB(TIME$,7,9)) ! SEED RANDOM # GEN
320 S = RND(-1*S)
330 PRINT 'READY?'
340 INPUT A$
350 IF A$ = 'NO' THEN STOP
360 !
370 !   BEGIN MAJOR LOOP
380 !
381 Z = 0
390 FOR W = 1 STEP 1 WHILE Z = 0
400   ! W IS PROBLEM # AND Z IS EXIT FLAG.
402   PRINT CHAR(140)
410   IF T = 1 THEN V = 1 ! V IS INDEX INTO SYMBOL ARRAY S$
420   IF T = 2 THEN V = 2
430   IF T = 3 THEN V = 3
440   IF T = 4 THEN V = FNA(3,1)
450   B = FNA(9,3)
460   IF V = 1 THEN DO
470     A = FNA(9,3)
480     Q = A + B
490   DOEND
500   IF V = 2 THEN DO
510     A = FNA(B,1)
520     Q = B - A
```

```
530   DOEND
540   IF V = 3 THEN DO
550     A = FNA(9,3)
560     Q = A*B
570   DOEND
580   PRINT B:S$(V):A:'=':
590   ENTER U,M,C
600   IF C <> Q THEN DO
610     IF M >= Ø THEN DO
620       PRINT 'WRONG':N$:'.'
630     DOEND
640     ELSE DO
650       PRINT 'YOU TOOK TOO LONG, TRY AGAIN.'
660     DOEND
670     IF V = 1 THEN PRINT B:S$(V):A:'=':A+B
680     IF V = 2 THEN PRINT B:S$(V):A:'=':B-A
690     IF V = 3 THEN PRINT B:S$(V):A:'=':B*A
700   DOEND
710   ELSE DO
720     D = INT(3*RND(Ø) + 1)
730     IF D = 1 THEN PRINT 'RIGHT,':N$:'.'
740     IF D = 2 THEN PRINT 'VERY GOOD.'
750     IF D = 3 THEN PRINT N$:', YOU GOT IT!'
760     IF D > 3 THEN PRINT 'CORRECT!'
770     PRINT 'YOU TOOK':M:'SECONDS TO GET THE ANSWER.'
780     R = R + 1
790   DOEND
800   PRINT 'MORE?'
810   INPUT A$
820   IF A$ = 'NO' THEN Z = 1
830 NEXT W
840 !
850 !   END OF MAJOR LOOP
860 !
870 PRINT 'YOU GOT':R:'OUT OF':W-1:'CORRECT.'
880 PRINT 'GOOD BYE.'
890 END

>RUN
MATH      FRI, JAN Ø5 1979          15:57:31


HELLO, WHO ARE YOU?
!LAURA
OK,  LAURA  I HAVE SOME MATH PROBLEMS FOR YOU.
WHICH TYPE OF PROBLEMS WOULD YOU LIKE?
1. ADDITION
2. SUBTRACTION
3. MULTIPLICATION
4. MIXED
TYPE 1, 2, 3, OR 4
!4
HOW MANY SECONDS SHALL I GIVE YOU TO ANSWER EACH PROBLEM?
```

```
!1Ø
READY?
!YES
6 + 6 = 12
VERY GOOD.
YOU TOOK 2 SECONDS TO GET THE ANSWER.
MORE?
!YES
4 + 6 = 1Ø
LAURA , YOU GOT IT!
YOU TOOK 1 SECONDS TO GET THE ANSWER.
MORE?
!YES
3 X 6 = 12
WRONG LAURA .
3 X 6 = 18
MORE?
!YES
7 + 4 = 9Ø
WRONG LAURA .
7 + 4 = 11
MORE?
!NO
YOU GOT 2 OUT OF 4 CORRECT.
GOOD BYE.
>QUIT
```

Sample Program 3:

TEXT HANDLING WITH STRING FUNCTIONS

```
100   REM FNA$ - SIMPLE TEXT JUSTIFICATION,  12-20-78
110
120   REM CALLING SEQUENCE:
130   REM     STRING = FNA$(INPUT_STRING, OUTPUT_STRING_LENGTH)
140
150
160  DEF FNA$(X$, L)
170  L2 = LEN(X$)
180  N = 0     ! N = # OF WORD DELIMITERS (SPACES)
190
200  FOR I2 = 1 TO L2     ! COUNT # OF WORDS
210       IF SUB(X$, I2) = ' ' THEN N = N + 1     ! IF SPACE, INCREMENT
220  NEXT I2
230
240  IF N = 0 THEN DO
250       A2$ = X$ + SPA(L - L2)     ! HANDLE ONE WORD CASE
260  DO END
270  ELSE DO
280      S1 = INT( (L-L2)/N ) + 1     ! # SPACES TO PUT BETWIXT EACH WORD
290      S2 = (L-L2) MOD N     ! # RESIDUAL SPACES
300      A2$ = ''     ! CLEAR OUT RESULT
310
320      FOR I2 = 1 TO L2     ! LOOP THRU INPUT STRING
330           IF SUB(X$, I2) <> ' ' THEN DO     ! HIT SPACE?
340                A2$ = A2$ + SUB(X$, I2)     ! NO, NORMAL CHARACTER
350           DO END
360           ELSE DO
370                A2$ = A2$ + SPA(S1)     ! HIT SPACE, PUT IN S1 SPACES.
380                IF S2 <> 0 THEN DO     ! ANY RESIDUAL SPACES?
390                         A2$ = A2$ + ' '     ! YES, PUT A SPACE HERE.
400                         S2 = S2 - 1
410                DO END
420           DO END
430      NEXT I2
440 DO END
450 FNA$ = A2$
460 FN END
470 REM
480 REM
490 REM -- This is the main driver for the text justification system
500 REM
510 PRINT 'Very dumb text justification system'
520 PRINT
530 INPUT 'Enter input file: ', F1$
540 DEFINE READ FILE #1 = F1$
550 INPUT 'Margin: ',L9
560 ON END #1 GOTO 740
570 PRINT
580 PRINT
```

```
590 S$ = ''     ! Clear out text input accumulator
600
610 IF LEN (S$) - 1 > L9 THEN DO  ! Have we filled up enough input text ?
620    S$ = SUB(S$, 1, LEN(S$)-1)     ! Yes, remove the blank on the end
630    FOR J = L9  STEP (-1)  UNTIL SUB(S$,J) = ' '     ! Find the last word
640    NEXT J
650    PRINT FNA$(SUB(S$, 1, J-1), L9)     ! Call the justification routine
660    S$ = SUB(S$, J+1, LEN(S$)) + ' '     ! Pickup unprocessed input
670 DO END     ! And continue.
680 ELSE DO     ! Come here when we are ready to do another read.
690    READ LINE #1, F1$     ! Get a line of input text.
700    S$ = S$ + CVT$$(F1$, 152) + ' '     ! Concat with the unprocessed text
710 DO END     ! And continue.
720 GOTO 610     ! Try again, folks.
730 REM -- End of file processing
740 PRINT S$     ! Print out unprocessed text (at bottom of paragraph)
750 PRINT
760 STOP

>TYPE XXX
Four score and seven years ago, our fathers brought forth
to this continent a new nation,
conceived in liberty, and dedicated to the proposition that
all men are created equal.
>RUN
JUST      WED, JAN 10 1979          09:48:12


Very dumb text justification system

Enter input file: XXX
Margin: 35


Four score and seven years ago,
our fathers brought forth to this
continent a new nation, conceived
in liberty, and dedicated to the
proposition that all men are
created equal.

STOP AT LINE 760
```

APPENDIX B

ASCII CHARACTER SET

The following is a list of the ASCII character set with the corresponding decimal equivalent and the meaning of each character.

Decimal Value (with parity on)	ASCII Character	Explanation
128		Null or fill character
129		Start of heading
13Ø		Start of text
131		End of text
132		End of transmission
133		Enquiry
134		Acknowledge
135		Bell
136		Backspace
137		Horizontal tab
138		Line feed
139		Vertical tab
14Ø		Form feed
141		Carriage return
142		Shift out
143		Shift in
144		Data link escape
145		Device control 1
146		Device control 2
147		Device control 3
148		Device control 4
149		Negative acknowledge
15Ø		Synchronous idle
151		End of transmission block
152		Cancel
153		End of medium
154		Substitute
155		Escape
156		File separator
157		Group separator
158		Record separator
159		Unit separator
16Ø		Space
161	!	Exclamation point
162	"	Double quotation mark
163	#	Number or pound sign
164	$	Dollar sign
165	%	Percent sign
166	&	Ampersand
167	'	Apostrophe

168	(	Opening (left) parenthesis
169	)	Closing (right) parenthesis
170	*	Asterisk
171	+	Plus
172	,	Comma
173	-	Hyphen or minus
174	.	Period or decimal point
175	/	Forward slant
176	Ø	Zero
177	1	One
178	2	Two
179	3	Three
180	4	Four
181	5	Five
182	6	Six
183	7	Seven
184	8	Eight
185	9	Nine
186	:	Colon
187	;	Semicolon
188	<	Left angle bracket (less than)
189	=	Equal sign
190	>	Right angle bracket (greater than)
191	?	Question mark
192	@	Commercial at sign
193	A	(193 through 218 are upper case characters)
194	B	
195	C	
196	D	
197	E	
198	F	
199	G	
200	H	
201	I	
202	J	
203	K	
204	L	
205	M	
206	N	
207	O	
208	P	
209	Q	
210	R	
211	S	
212	T	
213	U	
214	V	
215	W	
216	X	
217	Y	
218	Z	
219	[	Opening bracket
220	\	Backward slant

221	]	Closing bracket
222	^	Circumflex or up arrow
223	_	Underscore or backarrow
224	`	Grave accent
225	a	(225 through 250 are lower case characters)
226	b	
227	c	
228	d	
229	e	
230	f	
231	g	
232	h	
233	i	
234	j	
235	k	
236	l	
237	m	
238	n	
239	o	
240	p	
241	q	
242	r	
243	s	
244	t	
245	u	
246	v	
247	w	
248	x	
249	y	
250	z	
251	{	Opening (left) brace
252	|	Vertical line
253	}	Closing (right) brace
254	~	Tilde
255		Delete

APPENDIX C

RUN-TIME ERROR CODES

Code Number	Message
1	GOSUBS NESTED TOO DEEP
2	RETURN WITHOUT GOSUB
3	EXCESS SUBSCRIPT
4	TOO FEW SUBSCRIPTS
5	SUBSCRIPT OUT OF RANGE
6	ARRAY TOO LARGE
7	STORAGE SPACE EXCEEDED
8	BAD I-O UNIT
9	BAD FILE RECORD SIZE
10	DA RECORD SIZE ERROR
11	UNDEFINED I-O UNIT
12	WRITE ON READ ONLY FILE
13	END OF DATA
14	END OF FILE
15	FILE IN USE
16	NO UFD ATTACHED
17	DISK FULL
18	NO RIGHT TO FILE
19	ILLEGAL FILE NAME
20	FILE I-O ERROR
21	FILE NOT FOUND
22	INPUT DATA ERROR
23	VAL ARG NOT NUMERIC
24	BAD LINE NUMBER IN ASC LN FILE
25	ILLEGAL OPERATION ON SEGMENT DIRECTORY
26	READ AFTER WRITE ON SEQUENTIAL FILE
27	ILLEGAL OPERATION ON BINARY FILE
28	UNDEFINED MATRIX
29	ILLEGAL SEG DIR REFERENCE
30	ILLEGAL FILE TYPE FOR POSITION
31	ILLEGAL POSITION RECORD NUMBER
32	WRITE USING TO NON-ASCII FILE
33	PRINT USING STRING IN NUMERIC FORMAT
34	PRINT USING NUMERIC IN STRING FORMAT
35	PRINT USING FORMAT WITH NO EDIT FIELDS
36	BAD MARGIN SPECIFIER
37	MATRIX NOT SQUARE
38	MISMATCHED DIMENSIONS
39	OPERAND AND RESULT MUST BE DISTINCT
40	2 DIMENSIONAL MATRIX REQUIRED
41	INV MATRIX IS SINGULAR
42	MOD - SECOND ARGUMENT ZERO
43	EXPONENTIATION - BAD ARGUMENTS
44	SIN, COS - ARGUMENT RANGE ERROR
45	TAN - OVERFLOW

46	ASN, ACS - ARGUMENT RANGE ERROR
47	EXP - OVERFLOW
48	EXP - ARGUMENT TOO LARGE
49	LOG - ARGUMENT < = Ø
5Ø	SORT - ARGUMENT < Ø
51	EXPONENT OVERFLOW, UNDERFLOW
52	DIVISION BY ZERO
53	STORE FLOATING ERROR
54	REAL TO INTEGER CONVERSION ERROR
55	ON GOTO-GOSUB OVERRANGE ERROR
56	RECORD NOT FOUND
57	RECORD LOCKED
58	RECORD NOT LOCKED
59	KEY ALREADY EXISTS
6Ø	SEGMENT FILE IN USE
61	INCONSISTENT RECORD LENGTH
62	RECORD FILE FULL
63	KEY FILE FULL
64	IMPROPER FILE TYPE
65	PRIMARY KEY NOT SUPPLIED
66	ILLEGAL OPERATION ON UNIT Ø
67	FATAL MIDAS ERROR
68	Ø RAISED TO Ø OR A NEGATIVE POWER
69	CONSTANT ON LEFT SIDE OF ASSIGNMENT STATEMENT
7Ø	MIDAS CONCURRENCY ERROR

APPENDIX D

ADDITIONAL PRIMOS FEATURES

This appendix contains a glossary of useful PRIMOS terms, an introduction to the system EDITOR, an introduction to command files, and a brief discussion of the TERM command, with which terminal characteristics can be modified.

GLOSSARY OF PRIME CONCEPTS AND CONVENTIONS

The following is a glossary of concepts and conventions basic to Prime computers, the PRIMOS operating system, and the file system.

- abbreviation of PRIMOS commands

Only internal PRIMOS commands may be abbreviated.

- binary file

A translation of source file generated by a language translator (BASICV, COBOL, FTN, RPG).

- byte

8 bits; 1 ASCII character.

- CPU

Central Processor Unit (the Prime computer proper as distinct from peripheral devices or main memory).

- current directory

A temporary working directory explained in the discussion on Home vs Current Directory later in this section.

- directory

A file directory; a special kind of file containing a list of files and/or other directories, along with information on their characteristics and location. MFDs, UFDs, and subdirectories (sub-UFDs) are all directories. (Also see segment directory.)

- directory name

The file name of a directory.

- external command

A PRIMOS command existing as a runfile in the command directory (CMDNCØ). It is invoked by name, and executes in user address space. External commands print GO when starting, and cannot be abbreviated.

- file

An organized collection of information stored on a disk (or a peripheral storage medium such as tape). Each file has an identifying label called a filename.

- filename

A sequence of 32 or fewer characters which names a file or a directory. Within any directory, each filename is unique. Directory names and a filename may be combined into a pathname. Most commands accept a pathname wherever a filename is required.

Filenames may contain only the following characters:

A-Z, Ø-9, _ # $ - . *

The first character of a filename must not be numeric. On some devices underscore (_) prints as backarrow (<-).

- filename conventions

Prefixes indicate various types of files. These conventions are established by the compilers and loaders, or by common use, and not by PRIMOS itself.

B_filename	Binary (Object) file
C_filename	Command input file
L_filename	Listing file
M_filename	Load map file
O_filename	Command output file
filename	Source file or text file
*filename	SAVED (Executable) R-mode runfile
#filename	SAVED (Executable) V-mode runfile

- file-unit

A number between 1 and 63 ('77) assigned as a pseudonym to each open file by PRIMOS. This number may be given in place of a filename in certain commands, such as CLOSE. PRIMOS-level internal commands require octal values. Each user may have up to 16 file units open at the same time. Certain commands or activities use particular unit numbers by default:

PRIMOS assigned units	Octal	Decimal
INPUT, SLIST	1	1
LISTING	2	2
BINARY	3	3
AVAIL	5	5
COMINPUT	6	6
SEG's loadmap	13	11
COMOUTPUT	77	63
EDITOR	1,2	1,2
SORT	1-4	1-4
RUNOFF	1-3	1-3

- file protection keys

See keys, file protection.

- home directory

The user's main working directory, initially the login directory. A different directory may be selected with the ATTACH command.

- identity

The addressing mode plus its associated repertoire of computer instructions. Programs compiled in 32R or 64R mode execute in the R-identity; programs compiled in 64V mode execute in the V-identity. R-identity and V-identity are also called R-mode and V-mode.

- internal command

A command that executes in PRIMOS address space. Does not overwrite the user memory image. Internal commands can be abbreviated. See "abbreviation of PRIMOS commands".

- key, file protection

Specifies file protection, as in the PROTEC command.

Ø	No access
1	Read
2	Write
3	Read/Write
4	Delete and truncate
5	Delete, truncate and read
6	Delete, truncate and write
7	All rights

- LDEV

Logical disk device number as printed by the command STATUS DISKS. (See ldisk.)

- ldisk

A parameter to be replaced by the logical unit number (octal) of a disk volume. It is determined when the disk is brought up by a STARTUP or ADDISK command. Printed as LDEV by STATUS DISKS.

- logical disk

A disk volume that has been assigned a logical disk number by the operator or during system startup.

- MFD

The Master File Directory. A special directory that contains the names of the UFDs on a particular disk or partition. There is one MFD for each logical disk.

- nodename

Name of system on a network; assigned when local PRIMOS system is built or configured.

- number representations

xxxxx	Decimal
'xxxxx	Octal
$xxxxx	Hexadecimal

- object file

See binary file

- open

Active state of a file-unit. A command or program opens a file-unit in order to read or write it.

- output stream

Output from the computer that would usually be printed at a terminal during command execution, but which is written to a file if COMOUTPUT command was given.

- packname

See volume-name.

- page

A block of 1Ø24 16-bit words within a segment (512 words on Prime 3ØØ).

- partition

A portion [or all] of a multihead disk pack. Each partition is treated by PRIMOS as a separate physical device. Partitions are an integral number of heads in size, offset an even number of heads from the first head. A volume occupies a partition, and a "partition of a disk" and a "volume of files" are actually the same thing.

- pathname

A multi-part name which uniquely specifies a particular file (or directory) within a file system tree. A pathname (also called treename) gives a path from the disk volume, through directory and subdirectories, to a particular file or directory. See the discussion on Pathnames in this section.

- PDEV

Physical disk unit number as printed by STATUS DISKS. (See pdisk.)

- pdisk

A parameter to be replaced by a physical disk unit number. Needed only for operator commands.

- phantom user

A process running independently of a terminal, under the control of a command file.

- SEG

Prime's segmentation utility.

- segment

A 65,536-word block of address space.

- segment directory

A special form of directory used in direct-access file operations. Not to be confused with directory, which means "file directory".

- segno

Segment number.

- source file

A file containing programming language statements in the format required by the appropriate compiler or assembler.

- subdirectory

A directory that is in a UFD or another o subdirectory.

- sub-UFD

Same as subdirectory.

- treename

A synonym for pathname.

- UFD

A User File Directory, one of the Directories listed in the MFD of a volume. It may be used as a LOGIN name.

- unit

See file-unit.

- volume

A self-sufficient unit of disk storage, including an MFD, a disk record availability table, and associated files and directories. A volume may occupy a complete disk pack or be a partition within a multi-head disk pack.

- volume-name

A sequence of 6 or fewer characters labeling a volume. The name is assigned during formatting (by MAKE). The STATUS DISKS command uses this name in its DISK column to identify the disk.

- word

As a unit of address space, two bytes or 16 bits.

SETTING TERMINAL CHARACTERISTICS

Terminal characteristics may be set with the TERM command. These characteristics remain in effect until you reset them or until you log out. The commonly used TERM options are listed below. Typing TERM with no options returns the full list of TERM options available. The format is:

TERM options

The options are:

-ERASE character	Sets user's choice of erase character in place of the default, ".
-KILL character	Sets user's choice of kill character in place of default, ?.
-XOFF	Enables X-OFF/X-ON feature, which allows programs to halt without returning to PRIMOS command level. Programs may be resumed at point of halt by typing CONTROL-Q. Programs are halted by typing CONTROL-S. Also sets terminal to full duplex (default value.)
-NOXOFF	Disables X-OFF/X-ON feature (default).
-DISPLAY	Returns list of currently set TERM characters. Also displays current Duplex, Break and X-ON/X-OFF status.

EDITOR

Prime's text EDITOR can be used to create and edit text files and programs. It has two modes, INPUT and EDIT. Either blank line followed by a carriage return, or two CR's in a row, switches the EDITOR from one mode to the other. EDITOR is handy for preparing documents, reports, letters and programs, and for properly formatting them via the RUNOFF feature of PRIMOS. It is also useful for creating command files, discussed below. A complete description of all EDITOR and RUNOFF features can be found in the New User's Guide To EDITOR and RUNOFF.

Input Mode

INPUT mode is used for creating new files or programs or for adding more text to an existing file. It accepts lines of text that are entered into the system by a CR, as are all PRIMOS commands. To create a new file or program with the EDITOR, type:

ED

This automaticaly puts the EDITOR into INPUT mode.

Edit Mode

EDIT mode is used to modify the contents of a file or program file. There are over fifty commands available for use in EDIT mode. A good summary of these commands is provided in the PRIMOS Programmer's Companion. These commands allow you to move lines from one point to another in a file, to delete lines, to load in other files or parts of files, to format a file according to your particular needs (using RUNOFF commands) and to make line-specific or general changes in vocabulary, terms, etc. For BASIC/VM programmers, this is extremely handy when changing program variables, when moving lines of code, and when combining several programs.

To modify an existing file, type:

ED filename

The EDITOR opens the file, makes a copy of it in the current working directory, and switches into EDIT mode. All changes made to this 'work' file will not be incorporated into the disk copy until the file is 'FILEd', or saved, with the FILE command of the EDITOR. Separate copies of the 'work' file and the disk file may be maintained by FILEing the 'work' file under a name of its own. For example, to save an edited file under a name other than the original, type FILE, followed by a new filename:

FILE new-filename

Summary of Editor Commands

Command	Function
APPEND	Attaches specified text to end of current line.
BOTTOM	Positions pointer to bottom of file.
CHANGE/stRl/str2/	Replaces string 1(strl) with string 2(str2).
DELETE[n]	Deletes n lines including the current line.
DUNLOAD	Copies specified # of lines from work file to another file and deletes those lines from the work file.
FILE	Saves current work file to disk. Takes filename argument if current work file is not to overwrite disk file copy.
FIND string	Finds first line below current containing given string.
GMODIFY	String-oriented editing routine for individual lines.
INSERT newline	Inserts newline following current line. newline becomes current line.
LOAD filename	Copies contents of filename into work file under current line.
NEXT [n]	Moves the pointer n lines. Positive or negative values accepted.
OVERLAY string	Overlays given string over current line, starting in column one.
POINT n	Positions the pointer to line number n.
PRINT [n]	Prints n lines.
QUIT	Quits out of an EDITOR session to PRIMOS command level.
RETYPE string	Deletes current line and replaces it with string.
TOP	Repositions pointer to top of file.

UNLOAD	Similar to DUNLOAD but does not delete indicated lines from work file.
WHERE	Prints current line number.

Using the EDITOR For BASIC Programs

The EDITOR can easily be used to write BASIC programs, even though BASIC/VM has its own 'Editor' facility. The advantage of the PRIMOS EDITOR is the freedom to work and make changes without having to worry about line numbers and other BASIC constraints. Programs should be typed in upper-case letters, as only upper-case commands and statements are accepted by the BASIC compiler. Also, all syntax rules, as detailed in Section 4, should be followed. After a program has been FILEd, it can be numbered for use in the BASIC subsystem by using the PRIMOS NUMBER command. Full details on this command are found in The Reference Guide to PRIMOS Commands.

The NUMBER command requests the name of the file to be numbered or re-numbered, the name of the output file, which must differ from the original, the starting number and the increment value. For example, to number an Editor-created program starting with line number 10 and continuing in increments of 10, do the following:

```
OK, slist ex
GO
INPUT A
INPUT B
PRINT A*B
H=A/B
IF H<Ø THEN 1Ø
PRINT H
END

OK, number
GO
INTREENAME, OUTTREENAME, START, INCR,
ex ex3 1Ø 1Ø
OK, slist ex3
GO
  1Ø INPUT A
  2Ø INPUT B
  3Ø PRINT A*B
  4Ø H=A/B
  5Ø IF H<Ø THEN 1Ø
  6Ø PRINT H
  7Ø END
```

COMMAND INPUT FILES (COMINPUT)

```
COMINPUT    | pathname [funit] |
            | -CONTINUE        |
            | -END             |
            | -PAUSE           |
            | -START           |
            | -TTY             |
```

The COMINPUT command causes PRIMOS to take commands and data input from a specified file rather than from the user's terminal. Command files are usually created with the EDITOR.

filename — is the name of the file from which input is to be read.

funit — is the PRIMOS file unit number on which the input file is to be opened. If omitted, File Unit 6 is assumed.

options — Specify control flow. -TTY tells PRIMOS resume accepting input from the terminal. A command file should end with COMINPUT -TTY. The other options and their effects are discussed in Reference Guide, PRIMOS Commands, FDR3108.

Note

The COMINPUT command must be specified with at least one parameter. If CO is specified with a null parameter the message: NOT FOUND is printed at the terminal. Note also that the inclusion of CLOSE ALL in a COMINPUT file closes the file and causes the error message COMINP FILE EOF to be displayed.

Command input files are especially useful for repetitive processes such as compiling and loading a series of programs, building libraries, running production, and changing erase and kill characters at LOGIN time. BASIC/VM has its own COMINP command, which is similar in function to COMINPUT. The format and syntax are slightly different, as explained in Section 6.

Comments can be included in a command file at PRIMOS level by preceding each comment with the characters /*. The end of each comment is delimited by the characters /*.

COMMAND OUTPUT FILES (COMOUTPUT)

The COMOUTPUT command tells PRIMOS to send a copy of all terminal input and output to a specified file (called a 'comout' file), as well as (or instead of) to the user's terminal. The format is:

COMOUTPUT [treename] [option-1]...[option-n]

The options are described below. Logical combinations of options are permissable.

-CONTINUE Continues command output to a file. With the -CONTINUE option, subsequent terminal output is appended to the file specified by filename.

-END Stops command output to a file and closes the command output file unit.

-NTTY Turns off the terminal output, i.e., does not print or display responses to command lines, including the prompt OK,. Once -NTTY has been specified, terminal output is not turned on until -TTY is specified in a subsequent COMOUTPUT command.

-PAUSE Stops command output to filename. However, the command output file, filename, remains open.

-TTY turns on the terminal output.

Command output (comoutput) files are useful when the user wants to keep a record of terminal transactions. PRIMOS opens File Unit 17 and writes all command input and output responses to the file specified by filename.

Examples

```
OK, COMO OUTPUT
GO
...
```

This command line arranges for subsequent terminal output to be written to the file named OUTPUT. Commands are echoed, and responses continue to be displayed at the terminal. The file named OUTPUT is overwritten if it already exists.

```
...
OK, COMO -NTTY
...
```

Terminal output continues to the file named OUTPUT, but no terminal output is printed or displayed at the terminal.

```
...
OK, COMO -END
```

The file named OUTPUT is closed. Furthermore, since -TTY was not

specified following a -NTTY, terminal output will not be printed or displayed until the command line:

```
COMO -TTY
```

is issued

```
...
OK, COMO OUTPUT -C -P
```

The file named OUTPUT is opened and positioned to end-of-file, but no terminal output is sent to the file.

Remember that command output files are not closed by CLOSE ALL; they must be closed explicitly by COMO -END, CLOSE unit-number or CLOSE filename.

APPENDIX E

ADVANCED FILE HANDLING

CONTENTS

The information contained in this appendix is intended to supplement that presented in Section 8. File handling operations such as READs and WRITEs are covered in more detail, with special attention given to the default ASCII file type. Other information concerns access methods and file properties, altering the record size in DA and SA files, and truncation patterns observed in each file type. This information may be useful in deciding which file types to use in various data handling situations.

ACCESS METHODS

Data files can be accessed by one of four basic methods: Direct access, Sequential access, MIDAS and Segment directory (SEGDIR) access. The direct and sequential access methods are detailed below. MIDAS and SEGDIR protocols are more complex. More information on these access methods can be found in the Subroutine Reference Guide, and in Reference Guide, MIDAS.

Sequential Access Method

In sequential access, files are treated as a series of variable-length records. In sequential access, a file pointer is maintained to indicate the "current record" (the one that is involved in the current I/O operation). After each access, the pointer is moved to the next sequential record, which then becomes the current record. Thus, in order to reach the end of the file, each record in the file, beginning with the first, must be accessed in turn. It is not possible to skip records or back up to a previous record in sequential access files, except to return to the top of the file. This is known as "rewinding" the file pointer.

Sequential files are space-efficient since the records are only as long as the data they contain. However, random access to a particular record is time-consuming, since all the records between the current position and the desired position must be read.

Direct Access Method (DAM)

Files structured for direct access require an additional set of pointers which point to each record in the file. These pointers are automatically defined and maintained by the system, so the user needn't worry about them.

Direct access files must have fixed-length records. The record length may be increased or decreased from the default size of 60 words. The record length information is then stored in the header of the file for use by the file pointer.

Data retrieval is extremely flexible in direct access files. Any record in the file can be randomly positioned to by number. Records are numbered consecutively from the top of the file to the bottom, beginning with number 1. As data are added to the file, the number of records increases as necessary. Positioning is done internally by the system and involves counting the number of records (and therefore the number of characters) which must be bypassed before the desired record is reached. For example, if the record size is set to 40 words, (80 characters) and data from the third record are to be read, the pointer 'calculates' that 160 characters must be skipped before the third record can be read. When 160 characters have been bypassed, the pointer will be positioned to the beginning of the third record. The importance of fixed-length records in direct access is readily apparent.

DATA STORAGE PATTERNS

The following pages describe how data are stored in each file type.

Data Storage in ASCII and Binary Files

All files in BASIC/VM can be generally classified according to file type and access method. File type refers to the manner in which a file stores data. This is known as data storage. There are two major data storage methods: ASCII and Binary.

The main difference between ASCII and binary files is the way in which they code data for disk storage. The same datum written to both an ASCII and binary file is converted to a different format in each type of file. Subsequently, when displayed at the terminal, the data in the files will not appear identical.

ASCII: In ASCII files, all data, both numeric and string, are represented in ASCII character code, packed two characters per 16-bit word. Numeric values entered in decimal character format are converted to floating point internal format. When a file is read, the data values must be re-converted. String values are returned as the string characters which correspond to the stored ASCII codes. Numeric values are converted from floating point format to decimal representation, and are formatted according to any print format conventions specified, e.g., decimal points, dollar signs, etc.

The main feature of ASCII files is their ease of inspection, i.e., they can be SLISTed, or TYPEd at the terminal. They store data just like terminal output. Thus, their record storage patterns can be easily monitored for data integrity.

Binary: String data are stored in binary files as they are in ASCII files, i.e., in ASCII code. Numeric data are stored in internal machine format, i.e., four-word floating-point representation. There is no conversion to or from ASCII representation, so complete data accuracy is retained.

However, binary files cannot be accurately inspected through SLIST, TYPE or ED (the PRIMOS editor). Their storage patterns cannot be easily assessed by the user.

Data Storage in Each File Type

Within the general ASCII and binary file groups, files are further subdivided according to properties like record type, (variable or fixed-length) and access methods. In addition to these previously mentioned properties, distinctions can be made on the basis of intra-record data structure.

During file access, information is retrieved from a file one record at a time. Usually, specific data items must be retrieved from a record during a file READ. Therefore, there must be some way of internally marking where one item ends and the next one begins within a record. This is known as intra-record data structure.

Below is a description of the major features of each type of file, with the exception of SEGDIR and MIDAS features which are dealt with later in this section. These descriptions are intended to aid you in selecting the files best suited to your particular data storage and access requirements.

ASC files: are the default file type in BASIC/VM. They store data exactly like terminal output. Semi-colons, commas and colons force data to be written to an ASC file exactly as they would be output by a similar PRINT statement. Each item written to the file is separated from the next datum by the appropriate number of spaces forced by the indicated delimiter. Thus, spaces are the actual data delimiters in the intra-record structure of ASC files.

```
10 DEFINE FILE #1= 'SPACE'
15 READ A,B,C
20 DATA 20,21,22
25 WRITE #1, A,B,C
30 WRITE #1, A;B;C
35 WRITE #1, A:B:C
40 CLOSE #1
45 END
>TYPE SPACE
20                    21                    22
202122
20 21 22
```

ASC files have variable-length records. The default size of 60 words can be decreased or increased as necessary. In cases where items larger than 60 words are being stored, record-size should be enlarged appropriately.

ASCSEP files: are ASCII sequential files that use commas instead of spaces as internal data markers. Data written to an ASCSEP file can be read back in the same form as written. For example, if the following values are written to an ASCSEP file:

```
12,13,14
```

They will be stored and read back as indicated:

```
>TYPE SEP
12,13,14,15,
>READ #1,A
>PRINT A
12
```

It should be noted that string items containing commas will be fragmented when read back from the file. Commas, both verbatim and inserted, are interpreted as data delimiters. For example, if the val ue '$12,000' is written to an ASCSEP file it is stored as:

```
$12,000,
```

When read back, following occurs:

```
READ#1, A$
PRINT A$
$12
```

The data has obviously been fragmented. This problem can be remedied by using an alternate form of READ, READLINE. READLINE accepts the entire contents of one ASCII record, including commas, semicolons, colons and spaces, as one datum.

ASCSEP files are sequentially accessed, but they have fixed-length records. Unlike ASCII direct access files, which also have fixed-length records (discussed below), ASCSEP file records are not blank-padded on the right to fill out any space not occupied by data. Instead, the physical end of the records marked by a carriage return (CR). The delimiters used to affect record structure in ASC default files have no effect in ASCSEP files. Regardless of whether a WRITE statement is terminated by a comma, semicolon, colon or blank, the next sequential WRITE statement will write data to the next record.

ASCLN files: are ASCII sequential files with variable length records and inserted line-numbers. Commas are inserted as delimiters in ASCLN files, just as they are in ASCSEP files. Every record added to an ASCLN file is preceded by a line-number. Records are numbered in increments of 10, beginning with 10. When values are read back from the file, line numbers are stripped away.

The LIN# (unit) function (See Section 10) can be used to determine the actual line number at which the current I/O operation is being performed.

```
>DEFINE FILE #1 = 'LINES', ASCLN
>WRITE #1, 200,300,400
>WRITE #1, 34.56
>REWIND #1
>TYPE LINES
  10 200,300,400,
  20 34.56,
>READ #1, A
>PRINT LIN#(1)
10
>PRINT A
200
>CLOSE #1
```

ASCLN files are the only data files which may be edited at BASICV command level. Like a BASIC/VM program, they can be called to the foreground, LISTed, edited with BASICV commands and resequenced. ASCLN files are convenient for data that are continually being updated or otherwise modified.

ASCDA files: are ASCII direct access files. They have fixed-length records which are blank-padded to completely fill any record space not filled with data. Commas are inserted as data delimiters just as in ASCSEP files. A special file containing pointers to each record in the ASCDA file is maintained by the system for use in random access to data records.

The intra-record data structure is similar to that of an ASCSEP file. Extra spaces and 'blips' are usually output at the terminal due to the blank-padding factor. For example:

```
>10 DEFINE FILE #3 = 'DIR', ASCDA
>20 WRITE #3, 12,13,14
>30 WRITE #3, 123.45
>RUNNH
STOP AT LINE 30
>TYPE DIR
12,13,14,

123.45,
```

As in ASCLN files, the LIN #(unit) function can be used to determine the exact location of a record pointer in a direct access file.

BIN and BINDA files: of both types have fixed-length records. String data are stored as in ASCII files, i.e., in ASCII code; numeric data are stored in internal machine formats, i.e., four-word floating-point representation, ensuring complete data accuracy. Program execution is also expedited because less translation time is required during numeric data access.

Although BIN and BINDA files both have fixed-length records, BIN files are accessed sequentially, while BINDA files are accessed by the direct access method.

Data storage patterns in binary files cannot be accurately inspected at the terminal. If a binary file is TYPED or LISTED, the data may or may not be recognizable. Despite the inspection inconvenience, binary files are extremely useful for scientific computations requiring complete precision and data accuracy.

Any portion of a record not filled with data is zeroed out rather than blank-filled.

Writing to ASC Files

ASC files have special properties which distinguish them from others. Some of the properties which affect data written to ASC files are discussed below.

In ASC files only, successive WRITE statements can be forced to continue writing data to the same record until the record is filled. A line of data written to an ASC file can contain colon, comma, or semicolon delimiters between data items. Data will be stored exactly as if they had been output by a PRINT statement. Each delimiter affects data storage differently:

- a comma - causes item to be placed in next print zone.
- a semicolon - causes next item to be placed in next character position.
- a colon - causes next item to be placed one character position from the preceding item.

The following program writes data to an ASC file utilizing each delimiter. Each line output after TYPE ASCII represents the contents of a logical record. User input is underlined.

```
10 DEFINE FILE #1='ASCII'
20 READ A,B,C,D,E,F,G
25 DATA 22,23,24,25,26,27,28
30 WRITE #1,A,B,
35 WRITE #1,C:D:
40 WRITE #1,E,F;
45 WRITE #1,G
50 WRITE #1,A:B:
55 WRITE #1,C:D:
60 WRITE #1,E:F:
65 WRITE #1,G
70 WRITE #1,A,B,
75 WRITE #1, C:D:
80 WRITE  #1,E;F;
85 WRITE #1,G
>TYPE ASCII
22                    23                    24 25 26              272
8
22 23 24 25 26 27 28
22                    23                    24 25 262728
```

ACCOMODATING LARGE DATA ITEMS

Altering Record Size For SAM Files

Increasing the record size in a default ASCII file allows data items larger than 60 words to be stored in one record. If a string item in excess of 120 characters is written to a file with default record length, as much of the item as possible is written to the current record; the rest is written to the next record. Records are added as needed to accomodate this item.

```
>DEFINE FILE #1='T1',ASC,4
>WRITE #1,'HARRY G. DORK'
>TYPE T1
HARRY G.
 DORK
```

If the combined length of several numeric items being written to an ASC file exceeds the set record length, the way they will be stored depends on their individual lengths and on the delimiters which separate them. This example illustrates several ways in which large data items can be stored by varying the delimiters. Note that the record size has been set at 6 words.

```
>10 DEFINE FILE #1 = 'PLAIN',6
>20 READ A,B,C
```

```
>30 DATA 12,13456,7800000
>40 WRITE #1,A,B,C
>45 WRITE #1,A;B;C
>50 WRITE #1,A:B:C
>55 WRITE #1,A
>60 CLOSE #1
>65 END
>RUNNH
>TYPE PLAIN
12
13456
7800000
1213456
7800000
12 13456
7800000
12
```

Data items will not be truncated when written to an ASC file, even though the results of a file READ may create this impression.

ASCLN files: have the same properties as ASC default files in regard to storage patterns. In each ASCLN file record, four character positions are occupied by the inserted line numbers. This should be kept in mind when setting the record size for this type of file. Overlarge data items are treated in the same manner as discussed above.

Example:

```
>DEFINE FILE #1 = 'SMALL', ASCLN, 5
>WRITE #1, 'TOTALLY OUTRAGEOUS'
>TYPE SMALL
  10 TOTA
  20 LLY
  30 OUTR
  40 AGEO
  50 US,
```

ASCSEP files: differ from ASC and ASCLN files both in structure and in data storage. Records in an ASCSEP file are fixed-length. Commas are automatically inserted as data delimiters. Each one takes up one character position in the record. If a data item too large to fit in a single record is written to an ASCSEP file, it will be truncated.

```
>DEFINE FILE #1 = 'S1', ASCSEP, 5
>WRITE #1, 'TOTALLY OUTRAGEOUS'
>TYPE S1
TOTALLY OU
```

If several numeric items are written to the same file in a single WRITE statement, the length of each item relative to the record size will determine whether it will be stored intact or truncated. If the current record can accomodate only one item, the next item in the list

will be written to the next record. If it is too large to fit into this record, it will be truncated.

Delimiters occurring at the end of a WRITE statement have no effect on subsequent WRITE statements as they do in ASC files.

BIN files: maintain data quite differently from ASCII files. Although the exact nature of intra-record data storage is not readily known, the following example indicates that data items larger than the set record size are not truncated. Instead, they are stored in a manner which allows entire data items to be retrieved with a single READ, even if the datum exceeds the set record length.

```
>DEFINE FILE #1 = 'BINARY', BIN, 5
>WRITE #1,12345678910.32
>TYPE BINARY
   [{J="
>REWIND #1
>READ #1, A
>PRINT A
12345678910.32
>CLOSE #1
```

Altering Record Size in DA Files

The method used to retrieve data from a direct access file requires all the records in the file to have the same length. When records are being added to the file, the record size should be kept in mind. If data items written to an ASCDA record contain fewer characters than the maximum set by the record size, the data are padded internally with blanks until the record is entirely filled. In BINDA files, unused record space is zero filled. If, on the other hand, data in excess of the currently set record size are inadvertently written to a DA file, one of two things can occur:

- If the current record is empty, and the datum exceeds the record size, it will be truncated.
- If a series of items, e.g., A,B,C$, are included in a single WRITE statement, and C$ will not fit entirely into the remainder of the record, the pointer moves to the next sequential record and C$ is stored there. If C$ is larger than a single record, it will be truncated.

Example:

```
1Ø DEFINE FILE #1 = 'TRZ', ASCDA,8
2Ø WRITE #1,1234,5678,'MAGNIFICENT'
3Ø WRITE #1, 'I AM EIGHTEEN CHAR', 123,456
4Ø REWIND #1
>TYPE TRZ
1234,5678,
```

```
MAGNIFICENT,
I AM EIGHTEEN C,
123,456,
```

Writing Blank Lines to a File

If no variables or values are specified with WRITE, a blank record will be added to the file. This causes a blank line to occur in the output when the file is TYPEd.

Example:

```
>DEFINE FILE #1 = 'BL1'
>DEFINE FILE #2 = 'BL2', ASCSEP
>WRITE #1,12
>WRITE #2,12
>WRITE #1
>WRITE #2
>WRITE #1,13
>WRITE #2,13
>TYPE BL1
12

13
>TYPE BL2
12,

13,
```

Truncating a File

Writing data to a record that already contains data results in the over writing of the old data. If the file is CLOSEd immediately subsequent to the overwrite, the file will be truncated. The record just written to thus becomes the last record in the file.

Example:

```
>TYPE ASCSEP
TWAS THE NIGHT BEFORE CHRISTMAS,
AND ALL THROUGH THE HOUSE,
NOT A CREATURE WAS STIRRING,
NOT EVEN A MOUSE.,
>DEFINE FILE #1 = 'ASCSEP', ASCSEP
>READ #1, A$
>PRINT A$
TWAS THE NIGHT BEFORE CHRISTMAS
>WRITE #1, 'AND SANTA WAS BROKE'
>CLOSE #1
>TYPE ASCSEP
TWAS THE NIGHT BEFORE CHRISTMAS,
AND SANTA WAS BROKE,
```

Beyond this point, the original lines in the file have now been overwritten or truncated, as shown above.

READING ASCII FILES

READing Default ASCII Files

The special properties of ASCII default files affect the results of file READs in a manner worth explaining in some detail. These READ features are not applicable to other file types.

READ results vary with the data delimiters within each record, as explained earlier. The READ pointer looks for commas as end of data markers in all file types. Thus, it does not consider the ASC spacing delimiters to be delimiters. Because of the properties of ASC files, they are the only ones adversely affected by this feature.

If several numeric items are written to a file with various delimiters, as shown below:

```
10 DEFINE FILE #1= 'SPACE'
15 READ A,B,C
20 DATA 20,21,22
25 WRITE #1, A,B,C
30 WRITE #1, A;B;C
35 WRITE #1, A:B:C
40 CLOSE #1
45 END
>RUNNH
>TYPE SPACE
20                 21                 22
202122
20 21 22
```

The following may result when several READs are performed:

```
>DEFINE READ FILE #1 = 'SPACE'
>READ #1, A
>PRINT A
2Ø2122
>READ #1,B
>PRINT B
2Ø2122
>READ #1,A,B,C
END OF FILE  AT LINE  Ø

>REWIND #1
>READ #1,A,B,C
>PRINT A,B,C
2Ø2122              2Ø2122              2Ø2122
>READ #1,A
END OF FILE  AT LINE  Ø
```

READing With Numeric and String Variables

When ASC data items are READ into a string variable, e.g., READ #1, A$, all the values in the record, spaces included, are returned as one datum. The first comma reached marks the end of the datum; however if there are no verbatim commas in the data, a single string READ will return the entire record as a single datum. For example, if the values 12, 13 and 14 are written to an ASC file record, and numeric and string READs are done, the results are as follows:

```
>DEFINE FILE #1 = 'JUNK'
>WRITE #1, 12,13,14
>REWIND #1
>READ #1,A
>PRINT A
121314
>REWIND #1
>READ #1, A$
READ #1, A$
>PRINT A$
12                  13                  14
>TYPE JUNK
12                  13                  14
>
```

All spaces are discarded in a numeric READ because they are not considered numeric in value. Therefore, all numeric items are concatenated when READ into a numeric variable.

A record containing both string and numeric values with no commas between data items, (commas can be inserted by writing them to the file

like this: 12,',',13...) can be read in its entirety with a single string variable, but not with a single numeric variable.

Example:

```
>DEFINE FILE #1 = 'ASCII', ASC
>WRITE #1, 'SUGAR', 12.00
>WRITE #1, 'FLOUR', 5.00: 'COFFEE'
>REWIND #1
>TYPE ASCII
SUGAR                12
FLOUR                5 COFFEE
>READ #1,A
INPUT DATA ERROR AT LINE  0

>READ #1,A$
>PRINT A$
FLOUR                5 COFFEE
>REWIND #1
>READ #1,A$,B
INPUT DATA ERROR AT LINE  0

>REWIND #1
>READ #1,A$,B$
>PRINT A$
SUGAR                12
>PRINT B$
FLOUR                5 COFFEE
```

The INPUT DATA ERROR message indicates that a string value can not be read into the numeric variable specified.

READing From Other Sequential Files

Values from a record are READ into the given numeric or string variable(s) specified with the READ statement. For sequential files, the following READ properties are observed:

- If a record contains numeric data only, READ #unit, A returns the first numeric datum in the record, as delimited by the first comma.

- If a record contains string data only, READ #unit, A$ reads the first string value as delimited by the first comma in the record. If a string item itself contains a comma, it will be truncated at the position where the comma occurs.

- If the record contains both string and numeric data, READ #unit, A will return a numeric item only if it appears first in the record: READ #unit, A$ will return either numeric or string values, depending on which occurs first in the record.

- String data can only be read into string variables, .e.g, READ

#1, A$. Numeric data can be read into either numeric or string variables, e.g., either READ #1, A or READ #1,A$ will return a numeric value.

Example:

```
>DEFINE FILE #1 = 'ASCSEP', ASCSEP
>WRITE #1, 12
>REWIND #1
>READ #1, A
>PRINT A
12
>REWIND #1
>READ #1, A$
>PRINT A$
12
>WRITE #1, 'STRINGY'
>REWIND #1
>READ #1,A,B$
>PRINT A, B$
12                    STRINGY
>REWIND #1
>READ #1,A,B
INPUT DATA ERROR AT LINE  Ø

>REWIND #1

>READ #1,A
>READ #1,B$
>PRINT B$
STRINGY
```

READing ASCLN Files

Because ASCLN files insert line numbers in front of each record, it is easy to tell where the file pointer is positioned during any I/O operation. The system-provided function, LIN #(unit), can be used to to display the current record number.

```
>DEFINE FILE #1 = 'LINE1', ASCLN
>WRITE #1,'HI!'
>WRITE #1,'HOW'
>WRITE #1, 'ARE'
>WRITE #1, 'YOU!'
>REWIND #1
>TYPE LINE1
   1Ø HI!,
   2Ø HOW,
   3Ø ARE,
   4Ø YOU!,
>READ #1,A$
>PRINT LIN#(1)
```

```
10
>PRINT A$
HI!
>READ #1,B$,C$,D$
>PRINT LIN#(1)
40
>PRINT A$:B$:C$:D$
HI! HOW ARE YOU!
```

Summary of READs on Sequential Files

The following example shows how the properties of sequential files influence the results of simple file READs.

```
10 ON ERROR #1 GOTO 270
20 DEFINE FILE #1 = 'ASC'
30 DEFINE FILE  #2 = 'ASCSEP', ASCSEP
40 DEFINE FILE #3 = 'ASCLN', ASCLN
50 DEFINE FILE #4 = 'BINSAM', BIN
60 READ A,B,C$
70 DATA 123.45,48,'$100,000'
80 FOR I = 1 TO 4
90 WRITE #I, A,B,C$
100 REWIND #I
110 NEXT I
120 FOR N = 1 TO 4
130 PRINT 'NUMERIC READ FOR FILE ON UNIT #': N
140 PRINT
150 READ #N, A
160 PRINT A
170 REWIND #N
180 PRINT 'STRING READ FOR FILE ON UNIT #': N
190 PRINT
200 READ #N, A$
210 PRINT A$
220 PRINT
230 REWIND #N
240 NEXT N
250 CLOSE #1,2,3,4
260 END
270 PRINT ERR$(ERR)
280 GOTO 170

>TYPE ASC
123.45                48                    $100,000
>TYPE ASCSEP
123.45,48,$100,000,
>TYPE ASCLN
  10 123.45,48,$100,000,
>TYPE BINSAM
   {s3333`$100,000>
>RUNNH
NUMERIC READ FOR FILE ON UNIT # 1
```

```
INPUT DATA ERROR
STRING READ FOR FILE ON UNIT # 1

123.45                48                    $100

NUMERIC READ FOR FILE ON UNIT # 2

123.45
STRING READ FOR FILE ON UNIT # 2

123.45

NUMERIC READ FOR FILE ON UNIT # 3

123.45
STRING READ FOR FILE ON UNIT # 3

123.45

NUMERIC READ FOR FILE ON UNIT # 4

123.45
STRING READ FOR FILE ON UNIT # 4

 {s3333`$100,000
```

The INPUT DATA ERROR message appears because a numeric READ was attempted on an ASC file record containing both numeric and string data. Since this was expected, an error trap mechanism was included in the program. The ON ERROR statement (line 10) is further described below.

Note that a string READ of a binary file produces strange results. The numeric data values are not converted to decimal form when read with a string variable.

READ* With Default ASC Files

In default ASCII files, the READ* statement is most useful when records contain only string values. The variables for READing thus must be carefully chosen to avoid INPUT DATA ERROR messages. This requires some familiarity with the data being READ or otherwise manipulated.

The following example DEFINEs and WRITEs string data to several sequential files. Both types of READs, with and without stars (*), are performed to afford some comparison between the two forms of READs on various SAM files.

Example:

```
10 DEFINE FILE #1= 'ASC*'
```

```
20 DEFINE FILE #2 = 'SEP*', ASCSEP
30 DEFINE FILE #3 = 'LN*', ASCLN
40 DEFINE FILE #4 = 'BIN*', BIN
50 REWIND #1,2,3,4
60 READ A$,B$,C$,D$
70 DATA 'RED','WHITE','BLUE','PURPLE'
80 PRINT 'FIRST READ WITHOUT *'
85 PRINT
95 FOR N=1 TO 4
100 WRITE #N,A$,B$
105 WRITE #N,C$,D$
110 NEXT N
120 REWIND #1,2,3,4
135 FOR N= 1 TO 4
136 PRINT 'THIS IS FILE ON UNIT #': N
137 PRINT 'BEGIN READ WITHOUT *'
140 READ #N,A$
145 PRINT A$
150 PRINT
155 READ #N,B$
160 PRINT B$
162 PRINT
165 REWIND #N
166 PRINT 'NOW READ WITH *'
170 READ * #N, A$
175 PRINT A$
178 PRINT
180 READ * #N,B$
185 PRINT B$
190 PRINT
200 REWIND #N
201 PRINT 'END OF READ ON UNIT #':N
202 PRINT
205 NEXT N
220 CLOSE #1,2,3,4
230 PRINT 'END OF TEST'
240 END
>RUNNH
FIRST READ WITHOUT *

THIS IS FILE ON UNIT # 1
BEGIN READ WITHOUT *
RED                 WHITE

BLUE                PURPLE

NOW READ WITH *
RED                 WHITE

BLUE                PURPLE

END OF READ ON UNIT # 1
```

```
THIS IS FILE ON UNIT # 2
BEGIN READ WITHOUT *
RED

BLUE

NOW READ WITH *
RED

WHITE

END OF READ ON UNIT # 2

THIS IS FILE ON UNIT # 3
BEGIN READ WITHOUT *
RED

BLUE

NOW READ WITH *
RED

WHITE

END OF READ ON UNIT # 3

THIS IS FILE ON UNIT # 4
BEGIN READ WITHOUT *
RED

BLUE

NOW READ WITH *
RED

WHITE

END OF READ ON UNIT # 4

END OF TEST
>TYPE ASC*
RED                    WHITE
BLUE                   PURPLE
>TYPE SEP*
RED,WHITE,
BLUE,PURPLE,
>TYPE LN*
   1Ø RED,WHITE,
   2Ø BLUE,PURPLE,
>TYPE BIN*
REDWHITEBLUEPURPLE>
```

" (usage) 2-5

$ (hexadecimal number) D-4

' (octal number) D-4

* (in pathnames) 2-11

-CANCEL (SPOOL option) 2-21

-DEFER (SPOOL option) 2-21

-DISPLAY, TERM option 13-8

-ERASE, TERM option 13-8

-FORM (SPOOL option) 2-21

-KILL , TERM option 13-8

-LIST (SPOOL option) 2-20

-NOXOFF, TERM option 13-8

-XOFF, TERM option 13-8

.>SECTION 8

<*> (current disk) 2-11

> (in pathnames) 2-10

? (usage) 2-5

Abbreviations, command 2-2, D-1

Access methods 8-6, E-1

Access restrictions 8-4

Access, direct E-1

Access, file 3-14

Access, random E-2

Access, sequential E-1

Accessing PRIMOS 2-12

Accessing segment directories 8-22

Accessing sequential files 8-10

Accessing the system 2-12

ADD #unit, MIDAS 8-29

ADD statement 15-2

Adding data to MIDAS files 8-29

Adding matrices 9-8

ALTER command 7-2, 14-1

ALTER parameters 14-1

Altering record size E-7

Angle brackets, convention 2-2

APPEND, option (DEFINE) 8-3

Appendix B B-1

Appendix C, error codes C-1

Appendix D, Additional PRIMOS features D-1

Arithmetic operators 4-5

Array names 11-3

Array names, string 12-3

Arrays, definition 9-1

ASC data storage E-3

ASC files, READ* E-16

ASC files, reading E-11

ASC, type-code 15-5

ASCDA data storage E-5

ASCDA files E-5

ASCDA files, accessing 8-20

ASCDA, type-code 15-5

ASCII character set B-1

ASCII data E-2

ASCII files E-2

ASCII to decimal conversion 9-16, Appendix B

ASCLN data stroage E-4

ASCLN files, accessing 8-11

ASCLN files, reading E-14

ASCLN, type-code 15-5

ASCSEP data storage E-4

ASCSEP files, accessing 8-11

ASCSEP files, reading E-14

ASCSEP, type-code 15-5

Assigning directory passwords 2-14

Assigning file units 8-2

Assignment statement 5-2

ATTACH (PRIMOS command) 2-13, 13-3

ATTACH command (BASICV) 3-15, 14-2

Audience 1-1

Automatic logout 2-23

AVAIL command, PRIMOS 13-3

Backslash, usage 2-5

BASIC/VM commands, list 4-10

BASIC/VM commands, reference 14-1

BASIC/VM file types 8-1

BASIC/VM statements, list 4-11

BASIC/VM statements, reference 15-1

BASIC/VM, features 1-1

BASICV command 3-1

BASICV command (PRIMOS) 3-16, 13-3

BIN files E-5

BIN files, accessing 8-11

BIN files, reading E-14

BIN, type-code 15-5

Binary data E-2

Binary files E-2

Binary operators 11-6

BINDA files E-5

BINDA files, accessing 8-20

BINDA, type-code 15-5

Blank lines, writing E-10

Braces, convention 2-2

Brackets, convention 2-2

Branching in a program 6-2

Branching statements 6-1

BREAK command 7-6, 14-3

Break key 2-4

BREAK OFF command 14-3

BREAK ON command 14-3

Breakpoints 7-6

Byte, definition D-1

Call-by-reference functions 10-14

Call-by-value functions 10-14

Cancelling spool request 2-21

Caret, usage 2-5

CATALOG command 3-3, 14-3

Central processor unit, definition D-1

CHAIN statement 6-7, 15-2

Chain, directory 2-7

CHANGE statement 15-2

CHANGE statement 9-16, Appendix B

Changing directory names 2-19

Changing file names 2-19

Changing kill and erase characters D-7

Changing terminal characteristics D-7

Changing working directory 2-13

Characters, format 15-19

Characters, special terminal 2-5

Characters, string 15-21

CLEAR command 3-19, 14-4

CLOSE #unit statement E-10

CLOSE (MIDAS) statement, using 8-30

CLOSE statement 15-3

CLOSE statement, using 8-17

Closing DA files 8-21

Closing data files 8-17, E-10

Closing MIDAS files 8-30

CNAME (PRIMOS command) 2-19

CNAME command, PRIMOS 13-3

CNAME statement 15-3

Colons, delimiters 5-10

Column separators 5-8

Combining programs 3-10

COMINP command 6-4, 14-4

COMINP command, BASIC/VM D-11

COMINP statement 6-4, 15-3

COMINPUT (PRIMOS) 6-4

COMINPUT command, PRIMOS 13-4, D-11

Command abbreviations 2-2, D-1

Command files, PRIMOS D-11

Command format conventions 2-2

Command input files D-11

Command mode 3-17

Command output files D-11

Commands, BASIC/VM 14-1

Commas, delimiters 5-8

Comments 4-9

COMOUT command, PRIMOS 13-4

COMOUTPUT command, PRIMOS D-11

Compatability, BASIC/VM 1-4

COMPILE command 3-6, 14-4

Completing a work session 2-23

Concatenation operator 12-4

Concepts, glossary D-1

Conditional branching 6-8

Conditional statements 6-1

Conditionals, multi-branched 6-9

Conditionals, single-branched 6-8

Configuration, MIDAS file 8-25

Constants, definition 4-2

Contents of directories 2-15

CONTINUE command 7-8, 14-5

Control key 2-4

Control, program 6-1

CONTROL-P, usage 2-5

Controlling file access 2-22

Conventions, BASIC/VM statement 15-1

Conventions, command format 2-2

Conventions, filename D-2

Conventions, glossary D-1

Conventions, MIDAS files 8-27

Conventions, PRIMOS commands 13-1

Conversion, ASCII-decimal 9-16

CPU, definition D-1

CREATE (PRIMOS command) 2-14

CREATE command, PRIMOS 13-5

Creating files 3-1

Creating new directories 2-14

CREATK utility 8-24

CREATK utility, using 8-33

Current directory, definition 1-5, D-1

Current disk 2-10

CVT$$ function masks 10-10

DA files, altering record size E-9

DAM files, defining 8-18

Data access, random 8-19

Data file handling 8-1

Data file, matrix I/O 9-18

Data files 8-1

Data files, deleting 8-22

Data files, error trapping 8-16

Data files, opening 8-2

Data files, WRITE USING 8-9

Data files, writing to E-6

Data input/ouput 5-1

Data lists 5-2

Data output statements 5-7

Data retrieval E-2

DATA statement 5-2, 15-3

Data storage E-2

Data, input from terminal 5-4

Data, reference, numeric 11-1

Data, restoring 5-3

Data, truncation of E-7

Debugging 7-1

Debugging commands 7-6

Declaring arrays 9-2

DEF FN statement 15-4

Default files (ASC) E-3

Default printing 5-8

Default protection keys 2-22

Default record size 8-3

Deferring spool printing 2-21

DEFINE FILE #unit statement 15-4

DEFINE FILE statement 8-2

DEFINE FILE, MIDAS format 8-28

DEFINE SCRATCH FILE # statement 15-4

Defining arrays 9-2

Defining DAM files 8-18

Defining files 8-2

Defining matrices 9-2

Definitions D-1

Definitions, functions 10-17

DELETE (PRIMOS command) 2-15, 2-22

DELETE command 7-1, 14-5

DELETE command, PRIMOS 13-5

Deleting data files 8-22

Deleting directories 3-5

Deleting files 2-22, 3-13

Deleting lines 7-1

Deleting SEGDIR data files, figure 8-23

Delimiters, colons 5-9

Delimiters, commas 5-8

Delimiters, data E-3

Delimiters, semi-colons 5-9

Demo program, MIDAS 8-30

Demo program, MIDAS, description of 8-34

Demo program, running 8-37

DET function, reference 9-13

Determining file size 2-19

Device, see also disk

DIM statement 9-2, 15-6

Dimensioning arrays 9-2

Direct access files 8-18

Direct access files, writing to 8-19

Direct access method E-1

Direct access statements 8-18

Directories, creating 2-14

Directories, deleting 2-15

Directory chain 2-7

Directory name, definition 1-5, D-1

Directory names, changing 2-19

Directory operations 2-13

Directory structures 2-6

Directory, current, definition 1-5, D-1

Directory, definition 1-5, D-1

Directory, examining contents 2-15

Directory, home vs. current 2-10

Directory, home, definition D-3

Directory, segment, definition D-6

Directory, user file, definition D-6

Disk see also device

Disk, current 2-10

Disk, logical, definition D-4

Disk, physical, definition D-5

Disk, see also device

DO statement 15-6

DO-DOEND statement 6-9

Documents, related 2-1

Double-quote, usage 2-5

Edit mode, EDITOR D-8

Editing and debugging 7-1

Editing, simple 3-8

EDITOR commands, summary D-9

EDITOR, PRIMOS D-8

EDITOR, PRIMOS, for BASIC/VM programs D-10

Ellipsis, convention 2-2

End of file 8-16

END statement 6-2, 15-6

ENTER # statement 5-7, 15-7

ENTER statement 5-6, 15-7

Entering matrix values 9-17

ER! prompt 2-6

ERL variable 7-10, 15-16

ERR variable 15-16

ERR variable 7-10

ERR$(ERR) function 15-16

ERR$(x) function 7-10

Error codes, run-time C-1

Error location 7-10

ERROR OFF statement 15-7

Error trapping, DA files 8-21

Error trapping, data files 8-16

Error traps 7-10

Error traps, using 7-12

Evaluating logical expressions 11-9

Evaluation of expressions 4-7

Evaluation, string expressions 12-5

Examining directory contents 3-3

Examining file contents 2-20

Examples, conventions 2-2

EXECUTE command 3-6, 14-5

Execution errors 3-6

Execution errors, trapping 7-10

Exiting BASICV 3-13

Expression evaluation 11-7

Expressions, definition 4-1

Expressions, evaluation of 4-7

Expressions, numeric 11-3

Expressions, string 12-3

External branching 6-4

External command, definition D-1, D-2

EXTRACT command 7-2, 14-6

Extracting program lines 7-2

Features, ASCDA files E-5

Features, ASCII file E-3

Features, ASCLN E-4

Features, ASCSEP file E-4

Features, BASIC/VM 1-4

Features, BIN files E-6

Features, BINDA files E-6

File access, controlling 2-22

File access, remote 3-14

FILE command 3-5, 14-6

File contents, examining 2-20

File copies, obtaining 2-20

File directory, user, definition D-6

File handling 8-1, E-1

File handling, definition 8-1

File hierarchy 2-7

File Management System 8-1

File names, changing 2-19

File operations 2-19, 8-2

File protection keys, definition D-4

File size, determining 2-19
File structures 2-6
File structures, figure 8-7
File truncation E-10
File type-codes, table 8-5
File types 8-1
File types, PRIMOS, table 2-8
File types, table 15-5
File units 8-2
File, binary, definition D-1
File, definition D-2
File, object, definition D-4
File, source, definition D-6
File-unit, definition D-3
Filename 8-3
Filename conventions D-2
Filename, definition D-2
Files, ASC E-4
Files, ASCDA E-5
Files, ASCII E-2
Files, ASCLN E-3
Files, ASCSEP E-4
Files, BIN E-6
Files, Binary E-2
Files, BINDA E-6
Files, deleting 2-22
Files, direct access 8-18
Files, printing 2-20
Files, SCRATCH 8-17
Files, temporary 8-17
FMS, PRIMOS 8-1
FOR loops 15-8
FOR statement 15-7
FOR-NEXT loop nesting 15-9
FOR-NEXT statements 6-11
Forcing call-by-value 10-15
Foreground file, listing 3-3
Format characters 5-11
Format characters, numeric 5-11, 15-19
Format characters, string 15-21
Format strings, WRITE USING 8-9
Format, command, conventions 2-2
Function-I/O interaction 10-18
Functions, definition 4-4
Functions, numeric 10-1
Functions, string 10-8
Functions, string, table 10-9
Functions, user-defined 10-12
Glossary, concepts and conventions D-1

Glossary, concepts and conventions D-1

Glossary, PRIMOS terms D-1

GOSUB statement 6-3,15-9

GOTO statement 6-3, 15-10

Graph program, sample A-1

Halting a program 7-8

Hierarchy of files 2-7

Home directory, definition D-3

Home vs. current directory 2-10

Hyphen, convention 2-2

I/O-function interaction 10-18

Identifying error codes 7-10

IF conditional 6-2

IF statement 6-8, 15-10

IF structures 6-8, 15-10

Illegal matrix multiplication 9-11

Immediate mode 3-17

Indices, MIDAS 8-24

Initializing matrices 9-5

INPUT LINE statement 15-11

Input mode, EDITOR D-7

INPUT statement 5-5, 15-11

Input, data 5-1

Input, terminal 5-4

Input, timed 5-6

INPUTLINE statement 5-5

Inserting program breaks 7-6

Inserting program halts 7-8

INT function, example 10-4

Internal command, definition D-3

Internal subroutines 6-3

Intra-record structure E-2

INV matrix function 9-13

Inverting matrices -12

Job number, definition 2-13

Keys, file protection, definition D-4

Keys, MIDAS 8-24

Keys, special terminal 2-4

Language elements 4-1

LBPS command 14-6

LDEV, definition D-4

Ldisk, definition D-4

Legal and illegal variables, table 4-4

LENGTH command 7-5, 14-7

Length, record E-7

LET statement 5-2, 15-11

LIN #(unit) function E-4

LIN #(unit), using 8-20

LIN modifier 5-10

LIN#(unit), using with ASCLN files E-14

Line-numbered files E-4

Lines, renumbering 7-5

LIST command 3-4, 14-7

List of commands, BASIC/VM 4-10

List of statements, BASIC/VM 4-11

LISTF (PRIMOS command) 2-15

LISTF command, PRIMOS 13-5

Listing spool queue 2-20

Lists, data 5-2

LIST[NH] command 14-7

Literal string constants 4-1

LOAD command 3-10, 14-7

Locating errors 7-10

Locations, errors 7-11

Logging in 2-13

Logging out 2-23

Logical disk, definition D-4

Logical expressions, evaluation 11-9

Logical expressions, table 11-9

Logical operators 4-6, 11-6

LOGIN (PRIMOS command) 2-13

LOGIN command, PRIMOS 13-5

LOGOUT (PRIMOS command) 2-23

LOGOUT command, PRIMOS 13-6

Loop statements 6-11

Loops 6-1

Loops, conditional 6-12

Lower case convention 2-2

Manual organization 1-1

MARGIN OFF statement 15-11

MARGIN statement 5-17, 15-11

Markers, data, internal E-3

Masks, CVT$$ function 10-10

Master file directory, definition D-4

MAT (ZER,CON,IDN,NULL) statement 15-12

MAT INPUT * statement 9-17

MAT INPUT statement 9-17, 15-13

MAT INV statement 15-13

MAT PRINT 15-14

MAT PRINT statement 9-17

MAT READ #unit statement 9-18

MAT READ statement 9-15, 15-14

MAT statements 9-5, 15-12

MAT TRN statement 15-13

MAT WRITE # statement 15-14

MAT WRITE #unit statement 9-18

Math program, sample A-4

Matrices, definition 9-1

Matrix addition 9-8

Matrix concepts 9-5

Matrix determinants 9-13

Matrix dimensions 9-5

Matrix I/O 9-17

Matrix initialization 9-5

Matrix inversion 9-12

Matrix multiplication 9-10

Matrix multiplication, illegal 9-11

Matrix operations 9-5

Matrix subtraction 9-10

Matrix, input 9-17

Matrix, writing to file 9-18

MAX operator 11-5, 11-7

Merging programs 3-10

Methods, access 8-6

MFD, definition D-4

MIDAS access statements 8-26

MIDAS demo program 8-30

MIDAS demo, running 8-37

MIDAS file, configuration 8-25

MIDAS file, definition 8-24

MIDAS file, reading 8-28

MIDAS files, closing 8-30

MIDAS files, description of 8-11, 8-24

MIDAS files, opening 8-27

MIDAS files, removing data 8-30

MIDAS, rewinding pointer 8-28

MIDAS, type-code 15-5

MOD operator 11-5, 11-7

Mode, command 3-17

Mode, definition D-4

Mode, immediate 3-17

Mode, program-statement 3-17

Modes of operation 3-17

Modifiers 6-2

Modifying loops 6-11

Modifying program lines 7-2

Multi-branched conditionals 6-9

Multiple Index Data Access System 8-24

Multiplying matrices 9-10

Nesting, FOR-NEXT loops 15-9

NEW command 3-1, 14-7

New User's Guide To EDITOR And RUNOFF D-8

NEXT statement 15-15

Nodename, definition D-4

Non-foreground file, listing 3-3

Non-owner password 2-14

Non-owner rights 2-22

Non-owner status 2-14

NONOWN 2-15

NULL 2-16

Number representations D-4

Number, job, definition 2-13

Number, user 2-13

Numbers, statements 4-8

Numeric arrays 9-1, 11-3

Numeric constants 4-1, 11-1

Numeric data 11-1

Numeric expressions 11-3

Numeric format characters 5-11

Numeric format characters, table 15-19

Numeric functions 4-4, 10-1

Numeric operators, table 11-5

Numeric READs E-12

Numeric scalar variables 4-2, 11-2

Numeric subscripted variables 4-2

Numeric system functions 10-1

Numeric system functions, table 10-2

Object file, definition D-4

Obtaining file copies 2-20

OK, prompt 2-6

OLD command 3-1, 14-8

ON END statement, using 8-16

ON END-GOTO statement 15-16

ON ERROR statement 7-10

ON ERROR-GOTO statement 15-16

ON-GOSUB statement 6-11, 15-15

ON-GOTO statement 6-10, 15-15

One-dimensional arrays 9-1

Opening DAM files 8-18

Opening data files 8-2

Opening MIDAS files 8-27

Opening SAM files 8-8

Opening temporary files 8-17

Operands, definition 4-1

Operations, directory 2-13

Operations, file 2-19, 8-1

Operations, matrix 9-5

Operator priority 4-7

Operator priority, string 12-6

Operators 11-3

Operators, definition 4-5

Operators, relational 11-6

Operators, relational, string 12-5

Operators, string 12-4

Option, convention 2-2

Options, program 3-9

Ordinary pathname 2-7

Organization, manual 1-1

Output stream, definition D-5

Output, data 5-1, 5-7

Overview of Prime's BASIC/VM 1-1, 1-4

OWNER 2-15

Owner password 2-14

Owner rights 2-22

Owner status 2-14

Packname, definition D-5

Page, definition D-5

Parentheses, convention 2-2

Parity masks, table 10-10

Partition, definition D-5

PASSWD (PRIMOS command) 2-14

PASSWD command, PRIMOS 13-6

Password, non-owner 2-14

Password, owner 2-14

Passwords in pathnames 2-13

Passwords, assigning directory 2-14

Pathname vs. filename 2-10

Pathname, definition D-5

Pathname, ordinary 2-7

Pathname, relative 2-10

Pathnames 2-7

Pathnames with passwords 2-14

PAUSE command 7-8

PAUSE statement 15-17

PDEV, definition 1-9, D-5

Pdisk, definition D-5

Phantom user, definition D-5

Physical disk, definition 1-9, D-5

Pointer, rewinding 8-11

POSITION statement 15-17

Positioning MIDAS file pointer 8-28

Precedence, operators, of 11-7

PRIMOS command format 13-1

PRIMOS commands, reference to 13-1

PRIMOS EDITOR D-8

PRIMOS features D-1

PRIMOS file types, table 2-8

PRIMOS files 8-1

PRIMOS terms D-1

PRIMOS tree-structured file system 2-9

PRINT (LIN,TAB,SPA) statement 15-17

PRINT statement 5-6, 15-17

PRINT USING characters 5-11

PRINT USING statement 5-11, 15-18

Print zones 5-8

Printing files 2-20

Printing matrices 9-17

Printing on special forms 2-21

Priority, expression evaluation 11-7

Priority, operational 4-7

Priority, string operator 12-6

Program control 6-1

Program editing 7-1

Program execution, tracing 7-8

Program halts 7-8

Program length 7-5

Program, process of 3-9

Program-statement mode 3-17

Programs, sample A-1

Prompts, system 2-6

PROTEC (PRIMOS command) 2-22

PROTEC command, PRIMOS 13-6

Protection keys, default 2-22

PURGE command 3-13, 14-8

Question mark, usage 2-5

QUIT command 3-13, 14-8

Random data access 8-19

Reaching end of file 8-16

READ # statement 15-23

READ #unit statement 8-10, E-11

READ * # statement 15-23

READ LINE # statement 15-22

READ statement 5-2, 15-22

READ vs. READLINE 8-15

READ* statement 8-11

READ*, default files E-16

READ, option (DEFINE) 8-4

Reading ASC files E-11

Reading data into a matrix 9-15

Reading SAM files 8-10

Reading sequential files E-13

Reading with READ* 8-11

Reading, data files E-11

READLINE statement, using 8-14

READs after WRITEs 8-11

READ[KEY] statement 15-22

READ[KEY] statement, using 8-28

Record size 8-3

Record size, altering E-7

Records, default size 8-3

Records, fixed-length E-1

Records, overwriting E-10

Records, truncation E-7, E-10

Records, variable-length E-1

Recycling data 5-3

Redimensioning matrices 9-7

Reference, BASIC/VM commands 14-1

Reference, BASIC/VM statements 15-1

Reference, DET matrix function 9-13

Reference, numeric data 11-1

Reference, PRIMOS commands 13-1

Reference, string data 12-1

Related documents 2-1

Relational operators 4-5, 11-6, 12-5

Relative pathname 2-10

REM statement 15-23

Remarks 4-9

Remote file access 3-14

REMOVE statement 15-23

RENAME command 3-12, 14-9

Renaming files 3-12

Renumbering, program 7-5

REPLACE statement 15-24

REPLACE statement, using 8-22

Representations, number D-4

RESEQUENCE command 7-5, 14-9

RESTORE statement 5-3, 15-24

Restriction, APPEND 8-4

Restriction, READ 8-4

Restrictions, DEFINE 8-4

Return key 2-4

RETURN statement 6-3, 6-11, 15-24

REWIND (MIDAS) statement, using 8-30

REWIND statement 15-24

REWIND statement (MIDAS) 15-25

Rewinding pointer 8-11

Rights, non-owner 2-22

Rights, owner 2-22

RND function, example 10-4

Rules of precedence 11-7

RUN command 3-7, 14-10

Run-time error codes C-1

Run-time errors 3-7

Running a program 3-7

Running programs from PRIMOS 3-16

RUNOFF, PRIMOS D-8

Sample programs A-1

Saving files 3-5

Scalar multiplication, matrix 9-10

SCRATCH files 8-17

SEG, definition D-5, D-6

SEGDIR files 8-22

SEGDIR, type-code 15-5

Segment directories 8-22

Segment directory access 8-22

Segment directory, definition D-6

Segment, definition D-6

Segno, definition D-6

Semi-colons, delimiters 5-9

Sequential access method 8-6, E-1

Sequential access statements 8-8

Sequential files, altering record size E-7

Sequential files, closing 8-17

Sequential files, opening 8-8, E-6

Sequential files, reading 8-10

Sequential files, READing E-13

Sequential files, rewinding 8-10

Setting terminal characterestics D-7

Single branched conditionals 6-8

SIZE (PRIMOS command) 2-19

SIZE command, PRIMOS 13-7

Size, record 8-3

SLIST (PRIMOS command) 2-20

SLIST command, PRIMOS 13-7

Source code, translation 3-6

Source file, definition D-6

SPA modifier 5-10

Space separators 5-9

Spaces, convention 2-2

Special terminal characters 2-5

Special terminal keys 2-4

SPOOL (PRIMOS command) 2-20

SPOOL command, PRIMOS 13-7

SPOOL option -CANCEL 2-21

SPOOL option -DEFER 2-21

SPOOL option -FORM 2-21

SPOOL option -LIST 2-20

Spool printing, deferring 2-21

Spool queue, listing 2-20

Statement conventions 15-1

Statement modifiers 6-2

Statement numbers 4-8

Statement syntax 4-8

Statements, BASIC/VM 15-1

Statements, definition 4-8

Statements, editing 7-2

STATUS command, PRIMOS 13-7

STEP modifier 6-11

STOP statement 6-2, 15-25

Storage patterns, data E-2

Stream, output, definition D-5

String array names 12-3

String arrays 4-2, 9-1, 12-2

String constants 12-1

String data, reference 12-1

String expression evaluation 12-5

String expressions 12-3

String format characters 5-14

String format field characters 15-21

String functions 4-4

String operators 4-6, 12-4

String READs E-12

String scalar variables 4-2, 12-1

String subscripted variables 4-2

String system functions 10-8

String variable names 12-3

Structure, intra-record E-2

Structure, segment directory 8-22

Structures, directory 2-6

Structures, file 2-6, 8-7

Sub-UFD, definition D-6

Subdirectory, definition D-6

Subroutines, internal 6-3

Subscripted variables 11-2, 12-2

Subtracting matrices 9-10

Summary, PRIMOS EDITOR commands D-9

Syntax errors 3-6

Syntax, statement 4-8

System functions 10-2

System information, table 2-17

System prompts 2-6

TAB modifier 5-10

Table of numeric system functions 10-2

Table, file types 15-5

Table, matrix operations 9-6

Table, numeric format field characters 15-19

Table, numeric operators 11-5

Table, string format characters 15-21

Table, string functions 10-9

Table, type-codes 8-5

Table, variables 4-4

Template, MIDAS file 8-33

Temporary files, defining 8-17

TERM command, options D-7

TERM command, PRIMOS 13-8, D-7

Terminal characteristics, changing D-7

Terminal characters, special 2-5

Terminal input 5-4

Terminal keys, special 2-4

Terms, MIDAS 8-27

Text handling program, sample A-7

Timed terminal input 5-6

TRACE command 7-8

TRACE OFF command 14-10

TRACE ON command 14-10

Tracing execution 7-8

Translating source code 3-6

Transposing matrices 9-13

Trapping errors 7-10

Trapping errors in data files 8-16

Tree-structured file system, figure 2-9

Treename 2-7

Treename, definition D-6

TRN matrix function 9-13

Truncation, data E-7

Truncation, file E-10

Tutorial reference, BASIC 1-1

Two-dimensional arrays 9-2

TYPE command 3-3, 3-5, 14-10

Type-ahead 2-6

Type-code, definition 8-3

UFD, definition D-6

Unary operators 11-6

Unconditional statements 6-2

Underscore, usage 2-5

Unit, definition D-6

UNLESS modifier 6-2

UNTIL modifier 6-2, 6-12

UPDATE statement 15-25

UPDATE statement, using 8-29

Updating MIDAS files 8-29

Upper case convention 2-2

User file directory, definition D-6

User number 2-13

User, phantom, definition D-5

User-defined functions 10-12

USERS command, PRIMOS 13-8

Using EDITOR for BASIC/VM programs D-10

Using PRIMOS 2-12

Variable names 11-3

Variables, definition 4-2

Variables, string 12-1

Variables, table 4-4

Volume, definition D-6

Volume-name, definition D-6, D-7

WHILE modifier 6-2, 6-12

Word, definition D-7

Work session, completing 2-23

Working directory, changing 2-13

WRITE # USING statement 15-26

WRITE #unit statement 8-9, E-6

WRITE statement 15-25

WRITE USING # statement 15-26

WRITE USING, formats 8-9

Writing matrix to file 9-18

Writing to DAM files 8-19

Writing to data files E-6

Writing to MIDAS files 8-29

Writing to SAM files 8-8

Writing, ASC files E-6

Writing, ASCDA files E-9

Writing, ASCSEP files E-8

Writing, BIN files E-8

Writing, BINDA files E-9

\ (usage) 2-5

^ (usage) 2-5

_ (usage) 2-5