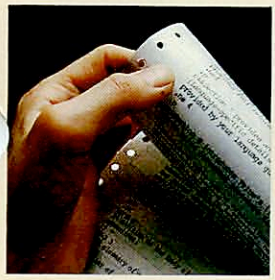
Prime Computer, Inc.

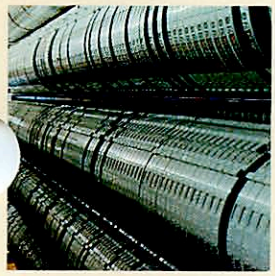
PRIME

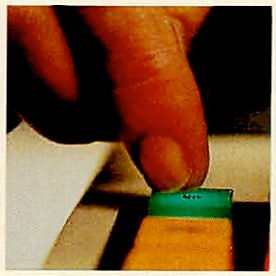
FDR3058-101B
Basic/VM
Programmer's Guide












BASIC/VM Programmer's Guide

by Laura J. Douros

with Update Pages for Rev. 19.0, July, 1982

by A. Paul Cioto

COPYRIGHT INFORMATION

PRINTING HISTORY – BASIC/VM Programmer's Guide

Edition	Date	Number	Documents Rev.
*First Edition	December 1977	IDR3058	14
*Update	July 1978	PTU2600-057	15
*Second Edition	April 1979	PDR3058	16.3
Third Edition	July 1980	FDR3058	17.2
Update 1	June 1981	COR3058-001	18.1
Update 2	July 1982	COR3058-002	19.0

*These editions are out of print.

HOW TO ORDER TECHNICAL DOCUMENTS

U.S. Customers

Software Distribution
Prime Computer, Inc.
1 New York Ave.
Framingham, MA 01701
(617) 879-2960 X2053, 2054

Customers Outside U.S.

Contact your local Prime subsidiary or distributor.

Prime Employees

Communications Services
MS 15-13, Prime Park
Natick, MA 01760
(617) 655-8000 X4837

INFORMATION Systems

Contact your Prime INFORMATION system dealer.

CONTENTS

PART V - REFERENCE

APPENDICES

I

OVERVIEW

1 Introduction

WHO NEEDS THIS MANUAL

The BASIC/VM Programmer's Guide is designed for the BASIC user or programmer who is acquainted with the BASIC language but who is unfamiliar with Prime's BASIC/VM. If you have never used the BASIC language, refer to commercial texts such as:

Marateck, Samuel, BASIC; Academic Press, Inc.

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

This guide defines Prime's BASIC/VM language and includes many examples of its usage. It also introduces Prime's operating system, PRIMOS, enabling new users to access and use the system without referring to other manuals.

DESCRIPTION OF BASIC/VM

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

The BASIC/VM compiler is an upward compatible extension of Prime's BASIC Interpreter, employing the fast program execution and virtual memory capabilities of the Prime 350 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 line-by-line modification of BASIC/VM programs.

Features

Below is a summary of the more important BASIC/VM performance features:

- Multiple users are supported without significant performance degradation.
- BASIC/VM is compiled, rather than interpreted, providing rapid program execution.
- Large programs may run without compromising small program efficiency.

- Multiple data segments and 128KB (64K word) procedure space accommodates large programs and arrays.
- All numeric data is double-precision and floating-point.
- A special set of functions handles all matrix operations.
- "Immediate mode" enables instant calculations.
- A special editor simplifies program modification.
- A library of numeric and string system functions simplifies many computations.
- Data output can be formatted with a variety of PRINT statement features, including cursor positioning control.
- All standard file types including MIDAS (keyed-index) can be accessed in BASIC/VM.
- CTRL-Ps and BREAKs can be trapped within the BASICV subsystem.
- Local variables and arrays are supported in user-defined functions.
- A "performance measurement" feature aids in debugging and optimizing code.

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 systems other than Prime's, read Section 2 to familiarize yourself with the terms and concepts of Prime's operating system. Additional information on PRIMOS terms and features, for example, EDITOR and FUTIL, are found in Appendix D.

If you have previously used a Prime computer and do not need a review of system access and PRIMOS terms, proceed to Section 3, which describes the important BASIC/VM commands and concepts.

A capsule summary of the BASIC/VM language elements, and a complete list of commands and statements, appears in Section 4. Programmers who have 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, is discussed in Sections 5 through 8. Section 9 deals with matrix and array manipulations. Section 10 contains a library of all numeric and string system functions and describes how to define and implement user-defined functions. Section 11 deals with the construction and evaluation of expressions. Sections 12-14 summarize the PRIMOS commands used in this guide, and all existing BASIC/VM commands and statements.

The Appendices contain a list of the BASIC/VM run-time error messages, the ASCII character set, a glossary of PRIMOS concepts and terms, an overview of useful PRIMOS features, like the EDITOR, and, several sample programs.

Compatibility with other forms of BASIC

While Prime's BASIC/VM is generally compatible with other versions of BASIC, users should note the following variations and special implementation features:

- The ability to enter several statements on one line is not supported.
- There is no "BYE" command; "QUIT" is its synonym in BASIC/VM.
- Both the double- and single-quote characters can be used as delimiters in BASIC/VM, for example, 'This is OK', or "This is OK". (The default PRIMOS erase character (") should be changed to another character before using double quotes as string delimiters in BASIC. See Section 3.)

18|

- Prime's BASIC/VM exclusively supports double-precision, floating-point numeric data.
- BASIC/VM supports the use of statement modifiers like WHILE, UNTIL, UNLESS with statements like IF, PRINT and FOR-NEXT loops.
- The assignment statement, "LET", is optional in BASIC/VM.
- BASIC/VM has control features like two-branch deciders, e.g., IF...THEN...ELSE; logical loop control via the modifiers WHILE, UNTIL, UNLESS; and DO...DOEND loops, all of which allow the writing of structured language.

2

Overview of PRIMOS

INTRODUCTION TO PRIMOS

This section is an overview of Primes's operating system, PRIMOS. It introduces all the commands BASIC/VM users will need to access and use the system. In addition, some background information on the file management system (FMS) and other important PRIMOS concepts are included. Appendix D of this guide supplements the information in this section with a glossary of PRIMOS terms and a brief synopsis of several PRIMOS utilities of interest to the BASIC/VM programmer.

Those users requiring additional information on all the subjects discussed in this section should consult the following Prime documents:

New User's Guide To Editor And Runoff
PRIMOS Programmer's Companion
Reference Guide, PRIMOS Commands
Subroutine Reference Guide

COMMAND FORMAT CONVENTIONS

The conventions for PRIMOS command documentation 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 appears in rust, the rust-colored 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 that 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.

CONVENTIONS IN EXAMPLES

In all examples, the user's input is **rust-colored**, and the system's output is not. For example:

```
OK, attach goudy
OK, ed seginfo
EDIT
```

User input usually may be either in lower case or in UPPER CASE. The rare exceptions will be specified in the commands where they occur.

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. (See **CONTROL-P**, **CONTROL-Q** and **CONTROL-S**, below.) Others are ignored.
- **RUBOUT** The key labeled RUBOUT has a special use in RUNOFF. It is not generally meaningful to other standard Prime software. On some terminals it is labeled DELETE or DEL.
- **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, or NEW-LINE.
- **BREAK, ATTN, INTRPT:** See **CONTROL-P**.

Special Characters

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

Backslash (/) Default EDITOR tab character.

Double-quote ("): Default erase character for PRIMOS Command Mode, EDITOR, and RUNOFF. 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 (described elsewhere in this guide) allows the user to choose a different erase character.

Question mark (?): Default kill character for PRIMOS Command Mode, EDITOR, and RUNOFF. Each question mark deletes all previous characters on the line. The PRIMOS command TERM 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.

CONTROL-S: Halt output to terminal, for inspection. Future input will not be echoed at the terminal until either CONTROL-P (QUIT) or CONTROL-Q (Continue) is given. This special function is activated by the command TERM -XOFF.

CONTROL-Q: Continue output to terminal following a CONTROL-S (if TERM -XOFF is in effect).

UNDERSCORE (_): On some devices, prints as a backarrow (←).

Note

The PRIMOS TERM command (described in Appendix D of this Guide) allows the user to choose a different ERASE and KILL character. If double-quotes are to be used as string

delimiters in BASIC/VM, a new ERASE character must be chosen. See Section 3 for details.

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. The punctuation mark following the "OK" indicates to the user whether he is interfacing with a single-user level of PRIMOS. The prompt "OK:" indicates single-user PRIMOS (a version of PRIMOS II); the prompt "OK," indicates multi-user PRIMOS.

PRIMOS and PRIMOS III support **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 jumbled with the command being typed ahead. Type-ahead is limited to the size of the terminal input buffer. Default is 192 characters.

PRIMOS II does not support **type-ahead**. The user must wait for "OK:" before entering the next command.

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

Changing the prompt message

PRIMOS also has a long form of prompt message which displays the time, the amounts of CPU time and I/O time used since the last prompt, and the user's stack level. (The stack level is only displayed if it's greater than 1. Most users don't need to worry about it.) Users can change the form of prompt message displayed at their terminal by giving the **RDY** command.

RDY -LONG	Sets the terminal to the long form of prompt.
RDY -BRIEF	Returns it to the standard "OK,".
RDY -OFF	Suppresses prompts entirely.
RDY -ON	Re-enables prompts.
RDY	Prints a single long-form prompt message.

For example:

```
OK, RDY -LONG
OK 09:21:29  0.284  0.324
RDY -OFF
RDY -ON
OK 09:21:43  0.036  0.000
RDY -BRIEF
OK,
```

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 description of the PRIMOS file system and a description of the ordering of information within files, refer to the **Subroutine Reference Guide.**

Pathnames

The PRIMOS file directory system is arranged as a tree. At the root are the disk volumes (also called paritions, or logical disks). Each disk volume has a Master File Directory (MFD) containing the names of several 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.

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, 0 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 0, until the UFD is found. For example, if there is no UFD named BEECH on disks 0, 1, or 2, then

```
BEECH>BRANCH5>SQUIRREL
```

will specify the same file as the previous two examples. This last form of pathname, in

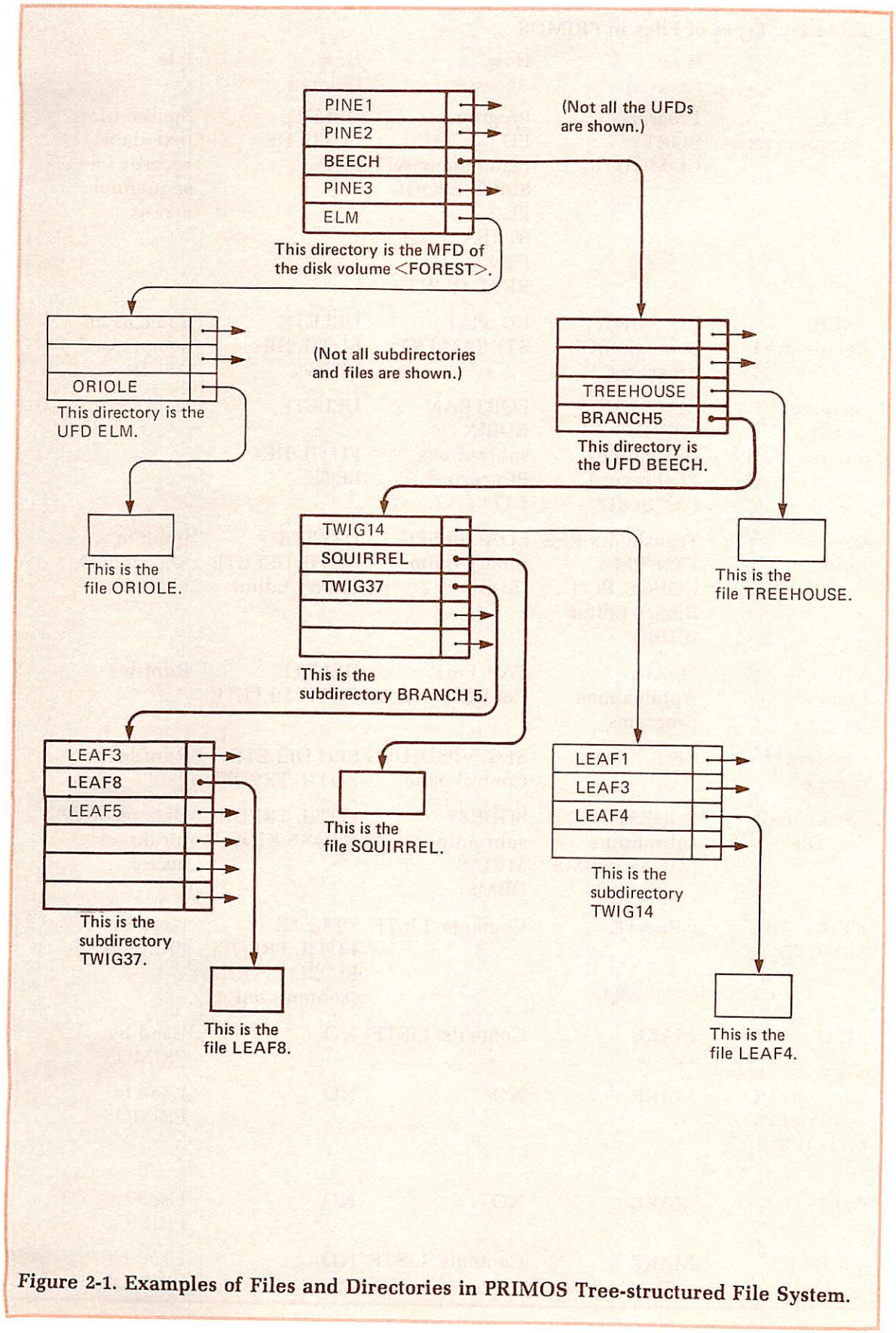


Figure 2-1. Examples of Files and Directories in PRIMOS Tree-structured File System.

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 PL/I STREAM I/D FTN READ/WRITE	DELETE FUTIL DE-LETE	Source files, text, data records for sequential access
ASCII, Compressed	ED, SORT, Some COBOL programs,	ED, PL/I STREAM I/O	DELETE FUTIL DE-LETE	Same as un-compressed ASCII
variable-length binary	FORTRAN WBRIN subroutines, PL/I record I/O, SORT	FORTRAN RDBIN subroutines PL/ record I/O	DELETE FUTIL DE-LETE	Data records
Object (Binary)	Translators:RPG, FTN, PMA, COBOL, PL/I Binary Editor (EDB)	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, DBG 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 FUTIL UFDDEL (contents only)	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
CMDNC0	MAKE	Contents: LISTF	NO	Used by PRIMOS

which the disk specifier is omitted, is called an **ordinary pathname** because it is very frequently used.

Pathnames vs filenames

Most commands accept a pathname to specify a file or a directory. So the terms "filename" and pathname may be used almost interchangeably. A few commands, however, require a filename, not a pathname. It is 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 the "*" alone, which means home directory.

Passwords

If any directory has a password, the password becomes part of the directory name or

pathname. Apostrophes are used to enclose the space between name and password (or to enclose the entire pathname, as the user wishes). For example, if the directory BEECH had a password, SECRET, a pathname using it might be

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

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 have 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 (CREATE; also see Editor, Appendix D).
- 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 working directory (work area), to files and to other system resource. 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 10'33 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.

During login, a misspelled UFD will cause the message "Not found. (LOGIN)" to be displayed. A misspelled or incorrect password will return the message "Insufficient access rights. (LOGIN)." If you get either of these messages, check to be sure you're logging into the right directory with the right password; then try logging in again. If you still have trouble, ask your supervisor for help. If the system itself is overloaded, a message such as "maximum number of users exceeded" may be displayed. In this case, log in again later, when some other user may have logged out.

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 (i.e., attach) to another directory in the PRIMOS tree structure with the ATTACH command. The format is:

A**TTACH 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:

A**TTACH '< 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 an 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.

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. For 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. Passwords may contain almost any characters; but they may not begin with a digit (0–9).

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**

If an attempt is made to delete directories containing files or subdirectories, PRIMOS prints the message:

```
The directory is not empty.  (DIRECTORY-NAME)
```

In this case, the user must do one of two things:

- Use the LISTF command to find what files (or subdirectories) are in the directory. Delete each entry with the command **DELETE filename.** Then delete the empty directory.
- Use FUTIL's TREDEL command (explained in appendix D) to delete files and directory simultaneously.

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=<MISCEL>TEKMAN>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 no files are 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.

FILE OPERATIONS

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

Table 2-2. Useful System Information

Item	Use	PRIMOS commands
Number of users	Indicates system resource usage and expected performance.	STATUS USERS (user list) USERS (number of users)
User login UFD	Identifies user who spooled text file (printed on banner).	STATUS, STATUS UNITS, STATUS ME
User number		STATUS ME, STATUS USERS
User line number		STATUS ME, STATUS USERS
User physical device		STATUS ME
Open file units	Avoids conflict when using files.	STATUS, STATUS UNITS
Magnetic tape units	Lists assigned units, with their logical aliases and users.	STATUS DEVICE
Disks in operation		STATUS, STATUS DISKS
Assigned peripheral devices	Tells what devices are available.	STATUS USERS
User priorities		STATUS USERS
Other user numbers		STATUS USERS
Your phantom user number	For logging out your phantoms.	STATUS USERS, STATUS ME
Network information	Tells if network is available.	STATUS, STATUS NET
Current nodename		STATUS NET, STATUS UNITS
Records available	Tells how much room is available for file building, sorting, etc.	AVAIL
System time and date	Performs time logging in audit files.	DATE
Computer time used since login	Measures program execution time.	TIME
Spool queue contents	Tells if job has been printed.	SPOOL -LIST
Names and status of printers	Tells if local printers are fuctioning.	PROP -STATUS
Environment for a printer	Gives parameters for printer's operations.	PROP printer-name -DISPLAY
Batch users	Identifies executing jobs, number of jobs per queue.	BATCH -DISPLAY

Table 2-2. (cont'd)

Item	Use	Primos commands
Your active Batch jobs	Gives job id, status Gives parameters	JOB -STATUS JOB -DISPLAY
Batch queue status		BATGEN -STATUS
Batch queue configurations	Shows environment of Batch system	BATGEN -DISPLAY

Note

Information given by any STATUS command is also given by the STATUS ALL command.

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. For example:

```
CREATE <1>TOPS>MIDDLE>BOTTOM
```

The sub-UFD BOTTOM is created in the sub-UFD MIDDLE, which in turn is found in the UFD TOPS, which is in the MFD of disk volume 1.

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.  DIRECTORY-NAME
ER!
```

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. For example:

```
cn tools>more_test oldtest
```

The file named MORE_TEST in the UFD TOOLS is changed to OLDTEST. Since no disk was specified, all MFDs (starting with logical disk 0) are searched for the UFD TOOLS.

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

```
Already exists.  OLDTEST
ER!
```

An incorrect old-name prompts the message:

```
Not found. MORETEST
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 440. However, a zero-word length file always contains one record. The format is:

SIZE **pathname**

Example:

```
OK, SIZE GLOSSARY
    14 RECORDS IN FILE
```

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. It is possible to suspend the terminal display as it is printing. See **Setting terminal characteristics,** elsewhere in this book.

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, PRTnnn, is x record[s] long.
```

nnn is a 3-digit number which identifies the file in the Spool Queue List. **x** is the number of records in the file. PRIMOS spools out short files as soon as possible; long files receive lower priorities.

Checking the queue : To check the status of the Spool Queue, give the command:

SPOOL -LIST

PRIMOS returns a list of all the files on the Queue which have not yet been printed. Additional information, such as the size, destination, the PRT number, any options, the form-type and the login-name of the user who spooled the file are also specified. For example:

```
OK, spool -list
[SPOOL rev 17.1]

 user   prt  time  name           size  opts/#   form   defer  at: CAROUSEL
------  ---  ----- -------------  ----  ------  ------ -----  ------------
ELLEN   001 16:42 CARLSON.REPFIL   40       2  WIDE
TEKMAN  002  9:19 COB#01            3       3         22:00  NEWTON
TEKMAN  003  9:20 COB#02            0          WHITE  18:00
SCELZA  005  9:20 TIME TABLE.MEM    4          WHITE
SCELZA  006  9:20 TIME TABLE.MEM    4          WHITE
SCELZA  007  9:20 TIME TABLE.MEM    4          WHITE
TEKMAN  008  9:21 GORK              6                 18:00  2

OK,
```

Cancelling a spool request: To cancel one or more spool requests, the command format is:

SPOOL -CANCEL [PRT]n-1 [,n-2...]

where **n-1** , **n-2** , etc., are the numbers of your spool files to be cancelled. For example:

```
OK, spool -cancel 47, 048, prt049
[SPOOL rev 17.0]
PRT047 has been cancelled.
PRT048 has been cancelled.
PRT049 has been cancelled.
```

Printing multiple copies: You can request several copies of one file by using the -COPIES option:

SPOOL **filename -COPIES n**

n is the number of copies desired.

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 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 format for **time** is HH [:] MM [AM/PM]. If AM or PM is given, HH:MM (the colon is optional) must be in 12-hour format (e.g., 1000 PM). Otherwise, *time* will be interpreted as 24-hour format (in which 2200 is 10:00 PM and 1000 is 10:00 AM).

Printing on special forms: Line printers traditionally use one of two types of paper— "wide" listing paper, on which most program listings appear, and 8-1/2 x11-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.

Changing the header: The **AS** option tells the spooler to print your file under a different name. The form is:

SPOOL **filename -AS alias**

The **alias** will appear on the header and in the SPOOL -LIST display.

Printing at specific locations: Networks with several printers often arrange to have the printers read each other's queues. It is therefore possible for a spool request to be printed at another location, perhaps many miles distant. To insure that a spool request is printed where you want it, use the**-AT** option:

SPOOL **filename -AT destination**

destination is a word of 16 letters or less. A list of available destination-names should be obtained from the System Administrator. (If a destination appears in the heading of the SPOOL -LIST display, for example AT:NEWTON, then that destination is the default destination for spool requests. If no destination follows "AT:", then no default has been established, and spool requests without destinations may be intercepted by any available printer.

Eliminating headers: To have files printed without header or trailer pages, use the -NOHEAD option:

SPOOL **filename** -NOHEAD

This option is particularly useful with preprinted forms, but if you're using this option in a multi-user environment, you will have to identify your own jobs.

Multiple options: Any or all of the above options, (except -CANCEL) may be used jointly in a single SPOOL command line. For example:

```
OK, spool o_17 -as ex.1 -at bldg.1 -defer 22:00
[SPOOL rev 17.0]
Your spool file,  PRT048, is      1 record  long.
```

This particular command requests that the file named "O–17" be printed at the "bldg.1" printer, under the alias of "EX.1", at 10 pm (22:00).

Printing multiple files: The CONCAT command concatenates files into a single file, which can then be printed via the SPOOL command. The format for CONCAT is:

CONCAT **new-file-name [-options]**

Options govern the format of the print-out and the disposition of the files. For details, see CONCAT in the **PRIMOS Commands Reference Guide.**

When you give the CONCAT command without options, CONCAT goes into input mode. It asks for the names of the files to be concatenated, and prints a colon prompt. Type the filenames, one per line. A null line (carriage return) signals the end of list. CONCAT then goes into command mode, and prints a right-angle prompt. You can then type a QUIT to end the session. (You can also type "INPUT" to return to input mod. For more information see the **PRIMOS Commands Reference Guide.**)

A sample session might be:

```
OK, concat triplet
[CONCAT Rev 17.0]

Enter filenames, one per line:
: first
: second
: third
: (CR)
> q

OK,
```

If the file TRIPLET already exists, CONCAT asks:

```
OK to modify old TRIPLET?
```

Answering NO returns you to PRIMOS command level. Answering YES prompts a second question:

```
Overwrite or append?
```

Answering OVERWRITE causes CONCAT to replace the old TRIPLET with a new one. Answering APPEND preserves the existing contents of TRIPLET and adds the new ones at its end.

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**

SEG runfiles cannot be deleted by this command. They must be deleted by SEG's own delete command (explained in Appendix D) or by FUTIL's TREDEL command (explained in Appendix D).

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 = 0).

The values and meanings of the access keys are:

key	Rights
0	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

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 to the owner and only read rights are given to the non-owner.

Note

The default protection keys associated with any newly created file or UFD are: 7 0. The owner is given all rights and the non-owner is given none. Default values for the PROTEC command, however, are: 0 0. Thus, the command PROTEC MYFILE denies all rights to owner and non-owner alike.

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 number 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 a 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).

II

BASIC FEATURES

3

Using BASIC/VM

INTRODUCTION

This section introduces the fundamental concepts and commands of BASIC/VM. It tells you nearly everything you need to use this version of BASIC.

ACCESSING BASIC/VM

Once you've logged into PRIMOS (see LOGIN, Section 2), you can access any of the subsystems available under PRIMOS. To enter the BASICV subsystem, (called "BASICV subsystem" for short) type BASICV. The system then responds with the message and prompt, as shown here:

```
OK, basicv
BASICV REV18.0
>
```

When the right angle bracket prompt appears (>), BASIC/VM is ready to accept commands. At this point, you can begin entering commands or program statements, or you can call up an existing file to work with. To call up an existing file, type OLD followed by the filename. To create a new file, type NEW, followed by the new filename. All input to BASIC/VM can be typed in uppercase or lowercase. However, it's not necessary to specify either the OLD or NEW command if you just want to execute a few commands or program statements. BASIC/VM will automatically create a temporary file called NONAME.BASIC if you don't issue the OLD or NEW command with a filename. You can save anything stored in this temporary file by typing FILE.

BASIC/VM Filenames

Two types of filenames may be used for BASIC/VM files: those that end in the ".BASIC" suffix and those that do not. Use of the .BASIC suffix is recommended because it makes program identification easier and it can be used with wildcards in CPL programs. (See **The CPL User's Guide** for details).

When given a **filename** argument that does not contain the .BASIC suffix, the commands BASICV, CHAIN, EXECUTE, LOAD and OLD will first look in your directory for a file with a name corresponding to **filename**.BASIC. If such a file does not exist, the commands then look for **filename** (without the .BASIC suffix) as supplied on the command line.

Be Careful

Be careful of creating files with names which are identical except for the .BASIC suffix: for example, TEST and TEST.BASIC. Here's why. Suppose you have two programs in your directory with those names. If you type the command:

EXECUTE TEST

BASICV will automatically look for, and in this case find, the program named TEST.BASIC, which isn't what you intended. Similarly, the other commands listed above would always perform their actions on TEST.BASIC even if you supplied TEST as the **filename** argument.

18

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) or **filename**.BASIC 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:

```
>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:

```
>new
NEW FILE NAME:  test
```

This new file, TEST.BASIC, becomes the foreground file. It remains in the foreground until an OLD file is called in or you decide to create another new file. If neither the OLD or NEW commands are given, the default foreground filename, NONAME.BASIC is used.

Correcting errors

Section 2 describes the default PRIMOS ERASE (") and KILL (?) characters which are used to correct any errors made during keyboard input. "Default" means that PRIMOS assumes these characters are being used for character erasure and line "kill", unless you change them. To use the double quote (") character as a string delimiter in BASIC, you will not choose a new ERASE character, as described below.

Setting a new ERASE character

The PRIMOS TERM command is used to set new ERASE and KILL characters. TERM, like all other PRIMOS commands, must be issued at PRIMOS command level. For example, to make the & (ampersand) character the new ERASE character, type:

```
TERM ERASE &
```

The ampersand character can now be used to erase characters, just as the " character usually does. "&" will remain the new ERASE character until you change it or until you log out.

The TERM command and its options are summarized in Appendix D, **Additional PRIMOS Features.**

USING BASICV COMMANDS

Once the preliminaries are out of the way, you can create a new file or work with an existing one. A "file" in BASIC/VM can be either a "data-file" or a "program-file". A program-file, as opposed to a data-file, is executable. "Data-file" usually refers to a collection or list of data which contains no code, and is therefore not executable. In this guide, the term "file" generally refers to both program- and data-files where the distinction is not important. When the term "program" is used, however, it refers specifically to an executable file. Remember, all saved programs are files, but not all files are programs.

Routine operations

Programming in BASIC/VM involves several routine operations. The following is a list of such operations and the BASIC/VM commands that perform them:

Command	Function
CATALOG	Display contents of current working directory.
LIST	Display contents of foreground file.
TYPE	Display contents of non-foreground file.
FILE	Save a NEW or modified file.
RUN	Run a foreground program.

COMPILE	Check for syntax errors; translate source program into executable machine language.
EXECUTE	Execute a compiled source program.
LOAD	Combine two or more files.
RENAME	Rename a foreground file.
PURGE	Remove files from a directory.
QUIT	Exit the BASICV subsystem.

Examining directory contents with 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]**

options are one or more of the following:

Option	Definition
DATE	Displays date and time the file was last modified.
PROTECTION	Gives owner or non-owner protection attributes (see PRIMOS, Section 2).
SIZE	Gives 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 shown in rust-colored letters.

The following example shows a typical CATALOG display for a directory. The first column displays the names of all files and sub-ufd's; the second, the number of records per file; the third, the file type; the fourth, the owner access rights; the fifth, the non-owner access rights; the sixth and seventh, the time and date the file was last modified.

```
>CATALOG A

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 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:

```
>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
>LISTNH
XYZ
```

Saving a new or modified file

A new file is entered into the system by typing in data or statements in proper BASIC/VM form. Section 4 explains all statement and 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 currently set ERASE and KILL characters to correct typing errors.

When the entire file 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 file has been FILEd, it becomes an OLD file. However, it remains in the foreground until you replace it with another.

Compiling the source code

In order for a program to be run or executed, the source code (i.e., the statements as entered from the terminal) 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 function. 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 run-time error messages, see Appendix C. Errors discovered during the COMPILE process can usually be corrected with the simple edit procedures discussed later in this section.

Executing a program

Once a program has been successfully COMPILEd, it can be executed with the EXECUTE command. EXECUTE accepts a pathname option; therefore, it can run either a foreground or non-foreground program. The format is:

EXECUTE **[pathname]**

If the **pathname** of an executable binary file is specified, the program will be executed immediately. If a file has not previously been compiled, EXECUTE will compile and then run it.

When the EXECUTE command is given without the pathname option, the currently compiled code in user memory is executed. If no code exists, BASICV displays: "STOP AT LINE 0". Remember, an executable binary version of a source program must exist before the program can be executed.

Run-time errors

The EXECUTE process displays errors that occur during run-time (execution-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, that is, 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. BASIC/VM provides debugging commands for detecting various sorts of execution control problems. See Section 7.

Running a program

Foreground source programs can also be executed with the RUN command. RUN combines both the 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.

Like EXECUTE, RUN displays both compile-time and run-time errors. RUN prints out the program name while EXECUTE does not. RUN will not execute the program if syntax errors are detected. RUN also has a no-header option, **NH**. Execution may begin at any 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.

Modifying a file

Errors detected by COMPILE, EXECUTE and RUN can be corrected with simple editing techniques. These techniques include deleting lines, inserting new lines, and retyping lines. More advanced editing procedures are covered in Section 7. The procedures discussed here enable 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 fundamental steps of BASIC/VM program creation are outlined in Figure 3-1. Only the basic features of program development are shown in this flow chart. Other program options are covered in the text.

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
BASICV REV17.Ø
NEW OR OLD: NEW SAMPLE


>1Ø DATA 12.1,34,78
>2Ø READ X,Y,Z,A
>3Ø PINT X,Y,Z
>4Ø END
>COMPILE
3Ø PINT X,Y,Z
  ^
INVALID WORD IN STATEMENT
>3Ø PRINT X,Y,Z
>COMPILE


>EXECUTE
END OF DATA  AT LINE  2Ø



>15 DATA 1Ø,2Ø,3Ø
```

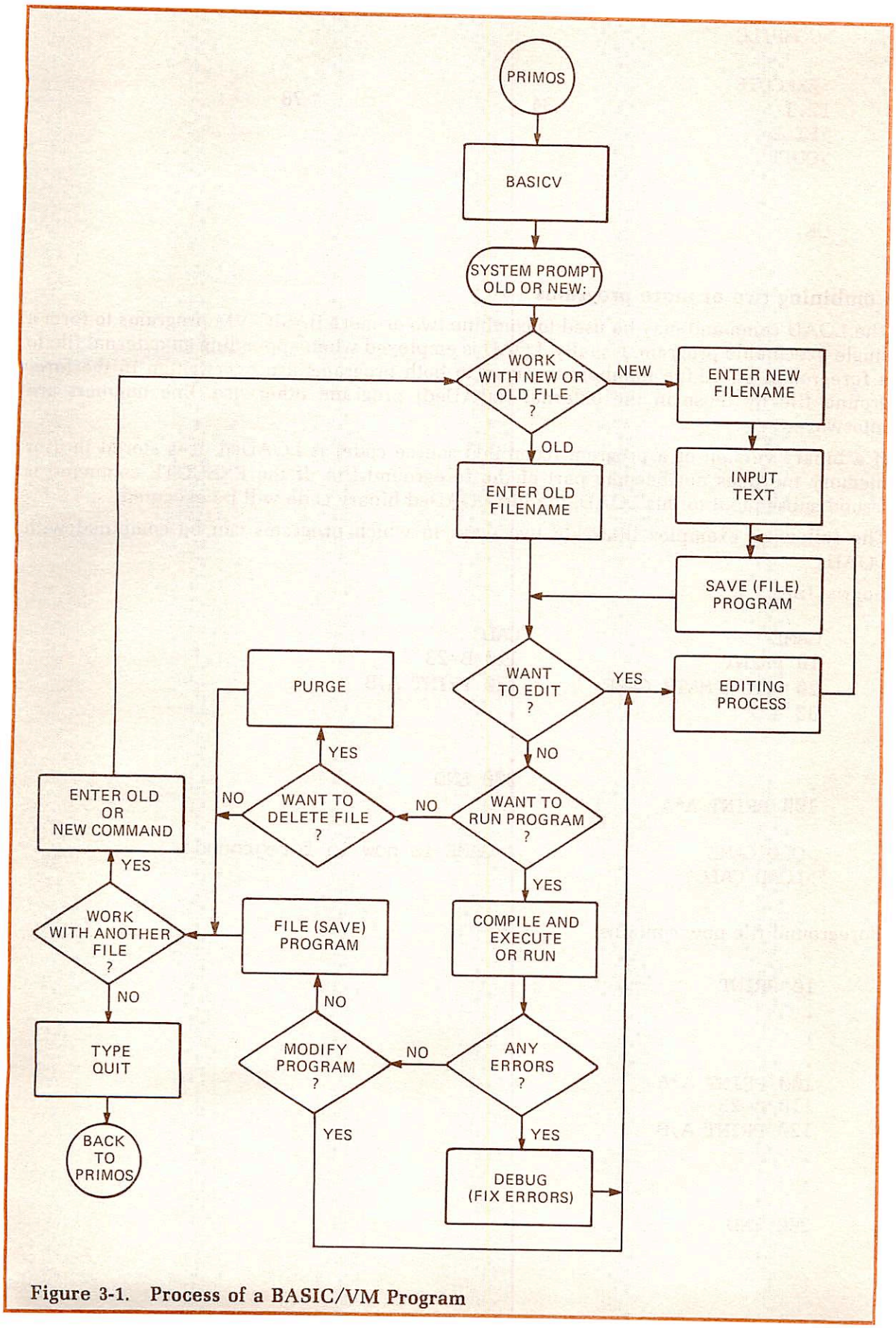


Figure 3-1. Process of a BASIC/VM Program

```
>COMPILE

>EXECUTE
12.1                    34                  78
>FILE
>QUIT


OK,
```

Combining two or more programs

The LOAD command may be used to combine two or more BASIC/VM programs to form a single executable program. Usually, LOAD is employed when appending an external file to a foreground one. Line numbers common to both programs are overwritten in the foreground file by those in the external (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 is issued subsequent 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.

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
```

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
35 C=11
40 PRINT A*B
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-filename** of the file need be specified. Note 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

To exit the BASICV subsystem and return to PRIMOS command level, use the QUIT command.

QUIT is issued at BASIC/VM command level and is the only way to return to PRIMOS command level, barring any access violations or disk I/O errors which may occur during program execution. This command closes any open files including those opened by BASIC/VM and PRIMOS), protecting them from accidental damage or modification. (QUIT does not close PRIMOS unit 63, the COMOUTPUT file unit. See Appendix D, **Additional PRIMOS Features**.)

ADDITIONAL INFORMATION

The following information deals with additional BASIC/VM features which may be of interest to some users. These features include:

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

RUNNING PROGRAMS FROM PRIMOS

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

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. Below is a BASIC program run from PRIMOS command level.

The following is a BASIC/VM program, called "T":

```
 5 J=Ø
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
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)

When input errors prevent the compiler from performing the requested action, an appropriate error message is displayed.

Command mode

Commands are directives to the BASICV system. If a command like "RUN" is issued in response to the > prompt, the compiler interprets it as an order to perform some action, and executes it immediately. More command-related information appears in Section 4.

Program-statement mode

Statements entered with preceding line numbers are immediately stored in user memory and are assumed to be part of a program unless otherwise indicated. These statements are not compiled or executed until the compiler is instructed to do so. For more details on program composition and statement syntax, see 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. Statements without line numbers are not stored in user memory and thus cannot become part of a program. Immediate mode is useful for debugging programs, for testing certain lines of code, and for performing quick calculations. Immediate mode is also referred to as "desk-calculator" mode.

Sample session

The following example depicts an immediate mode terminal session. User input is shown in rust-colored letters for clarity. (Note the absence of line numbers.)

```
>A=12

>B=25
>C=A+B
>PRINT C

37
>DIM A(3)
>A(1)=13
>A(2)=45
>A(3)=56
>PRINT A
12


>MAT PRINT A
13                    45                    56

>CLEAR
>PRINT A
Ø
>PRINT B


Ø
>PRINT C
```

```
Ø
>MAT PRINT A
UNDEFINED MATRIX AT LINE Ø
>QUIT
```

ACCESSING FILES IN REMOTE DIRECTORIES

Accessing files in BASIC/VM is similar to accessing files in PRIMOS. (See Section 2.) However, there are some important variations. The following examples illustrate BASIC/VM file access procedures for all possible file location situations.

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 on a remote disk

The general format of this procedure is:

OLD { **< volume >** | **< ldisk >** } **pathname**

The parameter **volume** is the name or number of the disk on which the file is stored **ldisk** is the name of the logical disk (see Appendix D for definition) which may be specified instead of a volume name or number. The angle brackets are required. **pathname** is the pathname of the file to be accessed.

For example, to access a file called GORDON listed in the UFD REV17 on a disk called SOFTWR, the following pathname would be typed:

```
OLD  <SOFTWR>REV17>GORDON
```

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

```
OLD  <3>REV17>GORDON
```

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

```
OLD <3>REV17 NEW>GORDON
```

In this case, the UFD REV17 is protected by the password NEW.

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 a sub-UFD under a UFD other than the current directory, the UFD name, and password, if any, must be specified. Example:

```
ATTACH LARRY>TEST
```

The UFD-name is LARRY and the sub-UFD-name is TEST.

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. The volume name of the remote disk, or a logical disk number, (ldisk) must be specified. For example, to attach to a sub-UFD called LILY listed under a passworded UFD called FLOWER on logical disk number 5, the format would be:

```
ATTACH '<5> FLOWER SNEEZE>LILY'
```

The password on FLOWER is "SNEEZE" and is required in the pathname.

4

Language elements

INTRODUCTION

The BASIC/VM language consists 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:

  Arrays
  Constants
  Functions
  Matrices
  Variables

- OPERATORS, of four types, which manipulate operands:

  Arithmetic
  Logical
  Relational
  String

All of these language elements are defined in this section. Additional information may be found in other sections of this manual as indicated.

BASIC/VM uses the full ASCII character set including:

- Letters from A-Z
- Digits from 0-9
- Special characters (see list in Appendix B)

OPERANDS

Within the context of a program, constants, functions, (or function references), variables and arrays, are all referred to as "operands". Operands are operated on, or manipulated, by operators, such as addition (+) or subtraction (–). Operators tell a program how to evaluate specific operands. A description of each type of BASIC/VM operand follows.

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: Numeric constants are positive or negative integers (whole numbers or decimal numbers), possibly in scientific notation. A numeric constant may have an optional sign (+ or –), a decimal point or an exponent specifier. If no decimal point is indicated in a number, BASIC/VM assumes it to be located immediately to the right of the right most digit.

All numeric data in BASIC/VM is double-precision and floating-point, with a level of accuracy to 13 significant figures in the mantissa and 3.5 significant figures in the exponent.

Some examples of numeric constants are:

8.88

-123

2.5E-2

Exponential notation: Numbers usually expressed in scientific notation are represented in "E notation", also known as "floating-point notation", in BASIC/VM. The general format for this representation is:

± **xx**E ± **nn**

xx is a whole or decimal number up to 13 digits in length; E is the exponent specifier (base 10) and **nn** is a 1- or 2-digit number representing a power of 10. The expression "10E6" means: "10 multiplied by 10 to the power of 6 or 10 times 10 to the sixth power".

As shown in the format, the plus (+) sign is optional: however, negative components must be preceded by a minus sign.

Exponential representation is quite flexible. For example, .001 can be expressed as 1E-3, or .01E-1, or 100E-5, to name a few. BASIC/VM automatically prints numbers with more than 13 digits in E notation. Exponent signs are printed after the exponent specifier, as in 1E-04, and 1E+04.

Literal string constants: String constants are a sequence of characters enclosed in quotes (') or double quotes ("). All spaces enclosed within the quotes are included in the string value. A "null" string contains no characters or spaces and is represented as '' or "". The maximum length of a string constant is 160 characters. The following are sample string constants:

```
'X,Y,Z'

"$123.56"

"Lou's sneakers"
```

Variables

Variables represent locations in memory where data values are to be stored. Values can be assigned with assignment statements, for example, A=12, or LET A=12, and as a result of calculations performed during program execution. 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: Numeric scalar variables consist of a single letter (A-Z) optionally followed by a single digit (0-9), for example, A6. The maximum length of a numeric variable name is two characters. There are 286 possible combinations of letters and numbers. Numeric scalar variables, also called "simple" numeric variables, are initialized to 0 by the BASICV subsystem before program execution. However, it is good programming practice to initialize all variables at the begining of a program.

A2
X4 } (numeric scalar variables)

String scalar variables: String scalar variables consist of a single letter (A-Z) followed by an optional digit, and a dollar sign($), for example, A$ or A2$. 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.

B$
A2$ } (string scalar variables)

Numeric subscripted variables: Numeric subscripted variables (also called "array elements") are simple numeric variables followed by one or two values enclosed in parentheses. These subscripted variables name elements in an array or matrix.

Arrays and matrices are generally visualized as having rows and columns, similar to a table. For instance, the subscripted variable, B(1,2), represents the element that exists in row 1, column 2 of array B. Arrays and matrices are defined by the DIM statement or a MAT statement. See Section 9 for details.

Referencing array elements: A singly-subscripted variable, for example, B(1), refers to an element in a one-dimensional array. In this case, "B" is known as an "array name." A variable with two subscripts, for example, C(1,2), refers to a particular element in the two-dimensional array named C.

In a doubly-subscripted variable, the first parenthesized value represents the row location, the second, the column location of a particular element in a two-dimensional array or matrix.

String subscripted variables: String subscripted variables are simple string variables followed by one or two values enclosed in parentheses. String array elements are represented by singly- or doubly-subscripted string variables. String arrays and matrices are dimensioned exactly as are numeric arrays and matrices. See Section 9 for details.

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

Numeric and string matrices: In BASIC/VM, a matrix consists of those elements of an array that have non-zero subscripts. For example, the array dimensioned by the statement, "DIM A(3)", has four elements: A(0), A(1), A(2), and A(3). Matrix A consists of only three elements: A(1), A(2) and A(3).

Avoiding name conflicts: A simple variable and an array may share the same name within a program, for example, A$. When an array element is referenced, subscripts are required; for example, the first element in the string array A$ is named by A$(1). When a scalar variable is referenced, no subscripts are used. Thus the use of subscripts distinguishes scalar variable from array references within the same program.

Both an array and simple scalar variable may have the same name, but the same array name may not be used for both a one- and two-dimensional array. For example, A(1) and A(1,2) cannot appear in the same program.

Local variables: The LOCAL statement can be used to declare certain variables and arrays **local** to the function definition in which they appear. This distinguishes them from **global** variables which appear outside of a function definition. Local variables and arrays are static, preserving their values over many calls to a particular function. Unlike global variables, they cannot be PRINTed in immediate mode or after a PAUSE or BREAK. See Section 10 for details.

Table 4-1. Legal and Illegal Variables

Type	Legal		Illegal	
Numeric scalar	A2	A	AB1	AR
	X4	Z	X14	BZ
String scalar	B$	B2$	AB$	AB3$
	A2$	A$	A21$	
Numeric subscripted	A2(1)	A(1,2)	A12(1)	A(1,2,1)
	A(1)	A2(1,2)	AB(1,2)	
String subscripted	A$(1)	A$(1,2)	A12$(1,2)	
	A2$(4)	A2$(1,2)	AB$(1)	

Functions

BASIC/VM provides a variety of numeric and string system functions such as TAN, COS, and LEN, to manipulate numeric and string data. Users may also define their own numeric and string functions, known as user-defined functions. See Section 10 for details.

Numeric functions: Numeric functions are identified by a three- or four-letter name, followed by parenthetically enclosed numeric items or parameters. Function definitions specify an operation, or a series of operations, to be performed on these listed items to produce a single value. Table 10-2 lists all available numeric functions. User-defined numeric functions are identified by the letters FN, followed by a simple numeric variable, as in FNA, FNA8.

String Functions: 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, to convert a numeric item to its corresponding string representation, and to represent a string item in ASCII code. All string system functions are listed in Table 10-2. User-defined string functions are named by the letters FN followed by a simple string variable, as in FNQ$.

OPERATORS

Within a program, operands are combined with operators to form "expressions". Operators indicate how the operands are to be evaluated. The four types of operators are arithmetic, relational, logical and string, and are listed below. The use of operators in evaluating expressions is detailed in Section 11.

Arithmetic Operators

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

Operator	Definition	Example
+	Addition (unary positive)	A+B, +A
–	Subtraction (unary negative)	A–B, –A
*	Multiplication	A*B
/	Division	A/B
^ or **	Exponentiation (involution)	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

Note

The operation commonly known as "exponentiation" is also referred to as "involution", according to the latest ANSI standard for BASIC. However, the term "exponentiation" is used exclusively throughout this book.

Relational operators

Relational operators are used with conditional statements and statement modifiers. (See Section 6 for details.) There are six relational operators:

Operators	Meaning	Example
<	Less than	A < B
>	Greater than	A > B
=	Equal	A$=B$
< = = <	Less than or equal	A$ < =C$
> = = >	Greater than or equal	A= > C
< > > <	Not equal	A < > D

String operators

String operands (items) may be combined with the concatenation operator (+) to form string expressions. Concatenation means "append one string to another". For example, if A$= "CORN"and B$="FLOWER", the expression, A$+B$ returns the string: "CORNFLOWER". String items may also be combined with relational operators to form string relational expressions. String operands may NOT be used with arithmetic (numeric) operators, nor may numeric operands be concatenated to string operands in the same expression. See Section 11 for details.

Logical operators

Logical operators are connectives for relational expressions, allowing the testing of many relations at once. The table below lists logical operators and their meanings.

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 made up of operands and operators and can be evaluated to produce a single result. Expressions can be numeric (arithmetic), for example, A+B; string, for example, C$+X$=Y$; relational, for example, A < =C+D; or logical, for example, A > C AND B=1. See Section 11 for details on expression composition and evaluation.

Evaluation

Expressions are evaluated according to operator priority. The priority list from highest to lowest for BASIC/VM appears 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

BASIC/VM system commands direct the BASICV subsystem to perform some immediate operation. Unlike statements, they are not preceded by line numbers. Some commands have optional parameters or arguments to further define the operation which they perform A list of all BASIC/VM commands appears at the end of this section. For complete BASIC/VM command format information, see Section 13.

STATEMENTS

When statements are included in a program, they are preceded by line numbers. When they are used without line numbers (in Immediate Mode), they are executed immediately. Example:

```
PRINT 12*154
1848
```

Results are obtained instantly, just as if you were using a calculator.

Statement syntax

Statements must adhere to the following rules:

- Each statement may be entered in either uppercase or lowercase letters.
- Each statement must be contained on one line.
- Statements must not exceed 160 ASCII characters in length.
- Portions of the statement (string literals) which are to be processed verbatim must be enclosed in quotes.
- Statements cannot be abbreviated.

Statements in a program should be separated from their identifying line numbers by 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, as this makes adding new statements easier. A statement may be added between lines 10 and 20, for example, without changing the numbers of the other statements. Given the following program:

```
10 PRINT 'NAME'
20 PRINT 'ADDRESS'
30 PRINT 'CITY'
>RUNNH
NAME
ADDRESS
CITY
```

A line may be inserted between lines 10 and 20 and another between 20 and 30, as indicated:

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

ADDRESS

CITY
```

The program now contains five statement lines instead of three.

Comments

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

```
10 REM  Remarks can appear like this-

20 ! or like this; or appended to a line as in:

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

LIST OF COMMANDS AND STATEMENTS

The tables briefly describe all available BASIC/VM system commands and language statements. Also included are references to other sections in this guide where more information on each command and statement may be found. Table 4-2 contains the list of commands, Table 4-3 the list of statements. Command abbreviations are in rust. 18

Table 4-2. List of Commands

Command	Description	Sections
ALTER	Allows editing of a single line in a program using the special subcommands listed in Table 13-1.	7,13
ATTACH	Changes location of working directory in BASIC subsystem; similar to PRIMOS ATTACH but works in BASIC environment.	3,13
BREAK {ON / OFF}	Sets and unsets breakpoints at specified line numbers for debugging. Maximum of 10 breakpoints may be set. See LBPS.	7,13
CATALOG	Lists all filenames under current UFD; optionally returns other file-related information.	3,13
CLEAR	Resets all previous numeric values to 0, all string values to null; deallocates any previously defined arrays and closes all open files.	5,13
COMINP	Calls a specified command file to foreground; reads and executes commands until a COMINP TTY is reached.	6,13
COMPILE	Translates a source file into executable binary form; displays syntax errors.	3,13
CONTINUE	Resumes program execution after a breakpoint or PAUSE.	7,13
DELETE	Deletes specified statement lines.	7,13
EXECUTE	Executes compiled code and displays run-time errors, if any.	3,13
EXTRACT	Deletes all except specified lines.	7,13

18

Command	Description	Sections
FILE	Saves all input and modifications to current (foreground) program file; writes file to disk.	3,13
LBPS	Lists currently set breakpoints.	7,13
LENGTH	Reports total number of lines in current program.	7,13
LIST[NH]	Displays the contents of current file at terminal. NH option suppresses header or program title.	3,13
LOAD	Merges or adds an external program to current (foreground) program.	3,13
NEW	Indicates to compiler that a new file is to be created in foreground. New filename must be specified.	3,13
OLD	Calls pre-existing file to current working area (foreground) and makes it current file.	3,13
18 PERF {ON, OFF, HIST, TABLE}	Turns performance measurement feature ON or OFF; issued prior to program compilation. HIST or TABLE options display measurement data in table or histogram form, respectively.	7,13
PURGE	Deletes specified file from UFD; file must first be closed.	3,13,14
QUIT	Returns control to PRIMOS from BASICV command level.	3,13
RENAME	Changes name of foreground file.	3,13
RESEQUENCE	Renumbers statements in current file.	7,13
RUN[NH]	Initiates compilation and execution of current source program.	3,13
TRACE {ON, OFF}	Tests program logic; line numbers are displayed as corresponding statements are executed.	7,13
TYPE	Displays contents of specified non-foreground file at terminal. TYPEd file does not replace file currently in the foreground.	3,13

Table 4-3. List of Statements

Statement	Description	Sections
ADD #	Adds record to MIDAS file opened on specified file unit.	8,14
18 CALL	Calls external declared subroutines.	6,14
CHAIN	Transfers program control to specified external program.	6,14
CHANGE	Converts ASCII character string to one-dimensional numeric array or vice-versa	10,14 APPB
CLOSE #	Closes files on specified unit(s).	8.14
COMINP	Stops execution of current program: calls specified command file to foreground.	6,13,14

Statement	Description	Sections
DATA	Contains numeric and string constants to be accessed by READ statement.	5,14
DEFINE FILE #	Opens a file of specified type on indicated unit number with optional access restrictions (APPEND, READ, SCRATCH).	8,14
DEF FN	Without FNEND, defines one-line function with numeric or string scalar variable arguments; with FNEND, defines multi-line function.	10,14
DIM	Defines dimensions of numeric or string array or matrix.	9,14
DO...DOEND	Defines a set of statements to be executed in association with IF-THEN statement pair.	6,14
ELSE DO	Optional alternative to DO-DOEND statement set.	6,14
END	Terminates program execution; no message is displayed.	6,14
ENTER [#]	A timed input statement: with # option, returns user-number assigned at LOGIN; also sets time limit on input.	5,14
FOR	Defines beginning and end of loop index; used optionally with STEP, WHILE, UNTIL, NEXT.	6,14
GOSUB	Transfers program control to internal subroutine: always used with RETURN.	6,14
GOTO	Transfers program control to specified line; can be used conditionally with IF or ON.	6,14
IF	Makes executable statements conditional; can be used with GOTO, THEN, ELSE DO, etc.	6,14
INPUT[LINE]	Requests data to be entered from terminal. LINE option accepts entire line, including commas and colons, as one entry.	5,14
LET	Assignment statement: optional.	5,14
LOCAL	Declares listed variable(s) or array name(s) to be "local" to the current function definition.	4,10
MARGIN [OFF]	Changes width of output lines. OFF option turns off all margin restrictions.	5,14
MAT { CON / IDN / NULL / ZER }	Sets initial value of matrix elements to zero, identity, null or one.	9,14
MAT { * / + / − }	Performs addition, subtraction or multiplication of two matrices.	9,14
MAT { INV / TRN }	Calculates INVERSE or TRANSPOSE values of one matrix and assigns them to another.	9,14
MAT INPUT	Reads data from a terminal and assigns values to elements of specified matrix or matrices.	9,14
MAT PRINT	Prints an entire matrix (or matrices) at terminal.	9,14

Statement	Description	Sections
MAT READ	Reads values from data list(s): assigns them to elements of matrix or matrices.	9,14
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.	8,9,14
MAT WRITE #	Writes an entire matrix (or matrices) to a file on specified unit.	8,9,14
NEXT	Defines end of loop begun by a FOR statement.	6,14
ON {GOSUB / GOTO} [ELSE GOTO]	Transfers program control to a subroutine (GOSUB) or to one of a list of statement numbers (GOTO). "ELSE GOTO" option routes control to another statement line if ON condition is unsatisfied.	6,14
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.	6,14
ON ERROR [#] GOTO	Defines statement line to which program control will transfer when a run-time error occurs, in disk file, if # specified.	7,8,14
ON QUIT GOTO	Traps QUITs generated by hitting CRTL-P or BREAK key during program execution; routes control to indicated line number.	6,14
PAUSE	Suspends program execution; to resume execution, type CONTINUE.	7,14
POSITION #	Positions internal record pointer to a specified record in DA disk file.	8,14
PRINT	Can be used with LIN,TAB,SPA options to print formatted data at terminal.	5,13,14
PRINT USING	Prints data output formatted according to format strings. See Table 14-2.	5,14
QUIT ERROR OFF	Turns off all QUIT traps previously set by ON QUIT GOTO statement.	6,14
RANDOMIZE	Used to reset random number generator (RND function) during or prior to program execution.	10,14
READ	Reads numeric or string values from a DATA statement in a program.	8,14
READ [KEY] #	Reads data associated with record key in MIDAS file opened on indicated unit.	8,14
READ LINE #	Reads an entire line of disk file text, including commas, and colons, as one data item.	8,14
READ [*] #	Reads from current record in file open on specified unit; pointer then positions to next sequential record. (* option holds pointer at current record after READ is complete.)	8,14
REM	Indicates a remark to the user.	4,14

Statement	Description	Sections
REMOVE #	Removes specific key from MIDAS file; if primary key, removes associated data record also.	8,14
REPLACE #	Deletes data files referenced by a segment directory. Moves pointer from deleted file to another file; zeroes old pointer.	8,14
RESTORE {#, $}	Reuses list of data items beginning with first item in lowest numbered DATA statement; # option reuses numeric items; $ option reuses string items only.	5,14
RETURN	Returns control from subroutine to statement following GOSUB statement.	5,14
REWIND #	Repositions record pointer to top of DAM or MIDAS file open on specified unit.	8,14
18| SUB FORTRAN	Declares subroutines observing the FORTRAN calling sequences for use from a BASIC/VM program.	6,14
WRITE #	Writes data to current record of disk file opened on specific unit.	8,14
WRITE USING #	Generates formatted output including tabs, spaces, column headings; writes output to ASCII disk file opened on specific unit.	8,14

III

PROGRAMMING IN BASIC/VM

5 Data I/O

INTRODUCTION

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

Statement	Description
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.
INPUTLINE	Accepts entire line of terminal input as one data item.
ENTER[#]	Timed input statement; # option assigns user login number to an indicated variable.

The second part deals with data output—the process of getting data from a program to a terminal or other output device. The following statements make it easy to obtain neatly formatted data output:

Statement	Description
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 character positions).

Data exchange involving the transfer of information between programs 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.

Multiple assignment

The multiple (simultaneous) assignment statement allows more than one variable to be assigned the same value. It also enables the use of "old" or previously assigned values in calculating a target value for a subscripted variable (array element). The general format is:

var-1, var-2 [,...,var-n] = expr

var-1, var-2, etc., are variables and/or array names; **expr** represents a string or numeric expression. A maximum of 100 variables may appear on the left side of the assignment statement. For example:

```
A, B, C = 12
```

All three variables, A, B and C, are assigned the value of 12.

Calculating subscript values

If an array element is referenced as a variable in a multiple assignment statement, BASIC/VM calculates the subscript value before assigning values to the rest of the variables listed. For instance:

```
I = 5
I, A(I) = 1Ø
```

The second assignment statement calculates A(I) to be equal to A(5), using the value for I from the previous assignment statement. The value of 10 is then assigned to A(5). Notice that the array-subscript calculation is performed first; the scalar variable I is then set equal to 10. Final result:

```
I = 1Ø
A(5) = 1Ø
```

For more information on arrays, see Section 9.

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.

```
10 data 5, 10, 15, 10
20 data 2, 7.2
30 read X, A1, A2, A3, B, C
40 print 'partial sum = '; A1+A2
50 print 'partial sum = ' ; A3+B+C
60 print 'average = ' ; (A1+A2+A3+B+C)/X
70 end
>runnh
partial sum = 25
partial sum = 19.2
average = 8.84
```

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

```
 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 one 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.

```
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
```

The program asks for three numbers. They should be separated by commas, colons or semicolons. If these separators 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.

Changing the default prompt: The default INPUT prompt (!) can be changed by including a prompt string in the following format:

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

Here is an example:

```
INPUT 'INPUT YOUR FAVORITE COLOR': C$
```

Alternatively, a PRINT statement may be used to print a prompt prior to the INPUT statement, as in the first example above.

Special characters in input: If commas are to be included as part of your input, use the INPUTLINE statement which accepts an entire line, including commas, colons, and semicolons as one entry. For example:

```
5 INPUTLINE 'WHAT IS YOUR FULL NAME:', A$
10 INPUTLINE 'WHAT IS YOUR OCCUPATION:', B$
15 PRINT
20 PRINT 'NAME', 'OCCUPATION'
25 PRINT
30 PRINT A$, B$
40 PRINT
45 PRINT 'THANK YOU'
50 END
```

Leading and trailing blanks may be included in string input by using single- or double-quote delimiters:

```
>INPUT 'INPUT YOUR FAVORITE COLOR':   C$
INPUT YOUR FAVORITE COLOR'      SALMON      '
>PRINT C$
      SALMON
```

The entry for C$ will be printed out exactly as entered, except for the quote delimiters which are discarded.

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; 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 numeric expression **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 to be assigned 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.

```
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).

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

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

The following short program uses all of the options of the ENTER # statement:

```
10 ENTER # T,5,H,P
20 PRINT 'YOUR USER NUMBER IS:':T
30 PRINT 'P=':P
40 PRINT 'YOUR RESPONSE TIME IN SECONDS WAS:':H
50 STOP
>RUNNH
!12
YOUR USER NUMBER IS: 28
P=12
YOUR RESPONSE TIME IN SECONDS WAS: 2
STOP AT LINE 50
```

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:

- Inserting blank lines in the output
- Separating data into columns (using commas)
- Spacing data items on a line (using colons or semicolons)
- Conditionally printing a line (using 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 or double quotes in a PRINT statement causes the string to be printed verbatim. Separating items by 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, and so forth. 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.

This program illustrates the use of commas to force data into columns:

```
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, semicolons and colons in PRINT statements. For 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
```

Using PRINT 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 numeric expression in parentheses representing the column number. (Negative arguments are treated as 0.) For example: | 19.0

```
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 numeric expression in parentheses representing the number of blank character positions. (Negative arguments are treated as 0.) For example: | 19.0

```
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
```

19.0| **LIN modifier:** A specific number of blank lines may be forced between items in one PRINT statement using the LIN modifier followed by a numeric expression in parentheses. This eliminates the need for consecutive PRINT statements. For 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(0) outputs a carriage return without a Line Feed.

Formatting with PRINT USING

Additional formatting capabilities are provided by the PRINT USING statement. 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 numeric or string format characters, depending on the type of data to be formatted.

There are seven special characters which define numeric format:

. , ^ + – $

Table 14-2 in Section 14 lists each character along with an example of its use.

The # sign: One or more pound signs (#) in a format string represent digit positions which are filled with the data provided in the statement.

```
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 instead of the item. This indicates that the item to be formatted was too large for the specified field. For example:

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

The period (.): The period character 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 (,): The comma character 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.

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

Results in the output:  2,000.00

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

    Results in the output:  +   30.6
```

The up arrow (^): The up arrow indicates exponentiation. Each ^ represents a character in the exponent field. Four up arrows indicate an exponent field which will be output as E±nn, where **nn** is a two-digit number whose value depends on how many places the decimal is moved in the mantissa.

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

Results in the output:  1735E-02
```

Plus and minus signs (+–): They are used to indicate the sign of an item 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.

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

Results in the output:  +25.00

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

Results in the output:  -12.30
```

One or more plus signs in a 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 as required by the size of the item.

```
PRINT USING '++##.##', 10.40

Results in the output:  +10.40

PRINT USING '++++.##', 15.90

Results in the output:  +15.90
```

One or more minus signs in a format have similar effect on data output. However, a positive datum will be preceded by a blank instead of a + sign.

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

     Results in the output:  20.00

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

     Results in the output:  -  705
```

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 14-3 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 output.

The pound sign (#): Each # sign in a 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-bracket (<): The primary function of the left angle-bracket is to left-justify text in a character string. In a string, a left bracket by itself causes the leftmost character to be printed:

```
>print using '<', 'UGANDA'
U
```

Other format characters in the format string dictate how many other characters will be printed, as well as the field positions they will occupy:

```
>print using '<##', 'UGANDA'
UGA
```

Right angle-bracket (>): The primary function of the right angle-bracket is to right-justify text in a character string. In a string, the right angle-bracket by itself causes the rightmost character to be printed in the first character position in the print zone:

```
>print using '>', 'UGANDA'
A
```

Depending on the other format string characters, the item may begin printing in the first character position of the print zone, or forced to other print positions as shown below:

```
>print using '>##########', 'YES'
        YES
```

Placing more than one left or right angle-bracket in a format string is not useful. All of the format characters following the second bracket are ignored:

```
>print using '<##<###', 'UGANDA'
UGA
```

A bracket that falls anywhere but the first character position is not useful. Everything is processed up to the bracket, and everything to the right of the bracket is ignored:

```
>print using '###>###', 'UGANDA'
UGA
```

The following program demonstrates several uses of the PRINT USING statement. User input is in rust-colored type 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 instead of the desired item, as shown below:

```
>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
19.0| of line length depends on the terminal and can be any number of characters from 1 to 1000.

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 margin checking, leaving a margin of infinite length.

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
```

The following results are obtained when the program is run:

```
1             2             3
4             5             6
7             8             9
10            11            12
```

This program sets the output line to 40 characters:

```
10 REM OUTPUT A MATRIX USING MAT PRINT
20 MARGIN 40
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
```

The output looks like this:

```
 1             2
 3             4
 5             6
 7             8
 9             10
 11            12
```

6

Program control statements

INTRODUCTION

Program control statements establish the order in which program statements are to be executed. Without such control indicators, program execution proceeds in ascending line number order. Specifically, control statements direct branching within a program, set up loops for repeated operations, transfer control to external programs, and tell a program when to stop.

Types of control statements

Control statements can be divided into two categories. If a statement directs control entirely within a program, it is termed an internal control statement: if it transfers execution control outside of a program to another program, it is an external control statement.

Types of internal control statements: There are three types of internal control statements: conditional, unconditional and loops. Conditional statements transfer program on the basis of a specified condition. This condition evaluates either to TRUE or FALSE. If a condition is true, one set of statements is executed; if false, an alternate path is taken. Unconditional statements affect execution control independently of any established conditions. Loop statements cause a program to loop or repeat a section of code until a specified condition is attained.

This section reflects the category division described above. The first part deals with control statements that direct the flow of execution entirely within a program; the last part of this section briefly discusses the control statements which surrender execution flow to external programs or command files.

UNCONDITIONAL CONTROL STATEMENTS

Control statements which unconditionally direct internal program flow are: GOSUB, GOTO, STOP and END.

Transfer to another statement

An unconditional 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. For example:

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

In this program segment, execution control is unconditionally transferred to line 50 from line 20.

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 the statement GOSUB, and program execution continues sequentially.

```
10 INPUT A
15 IF A<=0 GOTO 90
20 GOSUB 60
25 PRINT 'BACK TO LINE 25'
30 IF A<=10 GOTO 10
```

```
40 PRINT 'THE FINAL VALUE OF A IS:':A
50 GOTO 90
60 PRINT 'BEGINNING OF SUBROUTINE'
65 B=25
70 A=(A*A) +(A*B)
80 RETURN
90 STOP
>RUNNH
!2
BEGINNING OF SUBROUTINE
BACK TO LINE 25
THE FINAL VALUE OF A IS: 54
STOP AT LINE 90
>RUNNH
!-1
STOP AT LINE 90
```

The GOSUB statement in line 20 transfers control to the subroutine which begins at line 60. Line 80 returns control to line 25. Execution then continues sequentially until line 50, when control shifts to line 90 and the program STOPs.

Terminating program execution

The STOP and END statements terminate program execution. They do not cause branching or control transfer as such: they simply tell the program when to stop running. STOP prints out the message: **STOP AT LINE x**, where **x** is the appropriate line number in the program containing the STOP statement.

```
10 A=A+1
20 PRINT A
30 STOP
40 GOTO 10
>RUNNH
1
STOP AT LINE 30
```

The END statement acts like a STOP but prints no message. It is good programming practice to include one or the other of these statements in every program. This makes it easier to distinguish programmed execution halts from unscheduled ones.

```
10 A=A+1
20 PRINT A
30 END
>RUNNH
1
```

CONDITIONAL CONTROL STATEMENTS

Conditional statements generally operate in pairs or groups: the first statement in the pair sets a condition and the second statement provides an executable alternative depending on the value of this 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. The FOR statement is used with the NEXT statement to set up execution loops, or with other executable statements to establish execution conditions.

A condition is either true or false. If false, it has a value of 0; if true, it has a value not equal to 0. See Section 11 for details on logical expression evaluation.

Single condition loops

The FOR and NEXT statements together create a loop, or a series of statements that are executed repeatedly until a stated condition is met. The FOR statement begins the loop by initializing a variable, called an **index**, then setting a limit on its value. This forms the condition on which loop execution depends. The NEXT statement ends the loop and directs the program back to the FOR statement 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 executed until the **end** value for the index is reached. **incr**, a numeric expression, represents the increment value; the default is 1. For example: 19.0

```
10 PRINT 'X', 'X*2', 'X^2'
15 PRINT
20 FOR X=1 TO 5
30 PRINT X, X*2, X^2
40 NEXT X
50 END
>RUNNH
X                    X*2                  X^2

1                    2                    1
2                    4                    4
3                    6                    9
4                    8                    16
5                    10                   25
```

This program initializes the value of X to 1 in line 20, PRINTs the value, its double, and its square, returns control to line 20 and increments the value of X to 2. The loop continues until the value of X equals 5. When this happens, the program skips the "NEXT X" statement in line 40 and stops at line 50.

The default increment value for single condition FOR-loops is 1. To change this value, use STEP, followed by the desired increment value. For example, X can be incremented by 5 each time the "NEXT X" statement is executed:

```
10 PRINT 'X', 'X*2', 'X^2'
15 PRINT
20 FOR X=1 TO 20 STEP 5
30 PRINT X, X*2, X^2
40 NEXT X
50 END
>RUNNH
X                    X*2                  X^2
```

```
1          2          1
6          12         36
11         22         121
16         32         256
```

In this case, the value of X is initialized to 1, incremented to 6 after the first pass through the loop, and is incremented by 5 with each successive pass through the loop until X attains the value of 20.

Nesting loops

One or more loops may be placed within another, or "nested", in a program. Nesting must be done so that the inner loop terminates before the outer loop which contains it terminates. When two or more loops are nested, the last in a series of FOR-loops to be defined must have the first corresponding NEXT statement in the program. The first, or outermost, FOR-loop must have the last NEXT statement in the series. The program below shows how a series of loops should be nested:

```
 5 J,I,K=0  !Initialize index variables
10  FOR J=1 TO 2
20   FOR I =1 TO 3 STEP 1
30    FOR K=1 TO 4
40    PRINT (J+K)/I
50    NEXT K
60   NEXT I
70  NEXT J
80 END
```

In contrast, the program below shows improper FOR-loop nesting:

```
 5 REM THIS IS AN IMPROPERLY NESTED FOR-LOOP
10 DIM A(2,2)
20 FOR I=1 TO 2
30  FOR J=1 TO 2
40  A(I,J)=I*J
45  PRINT A(I,J)
50  NEXT I
55 NEXT J
60 STOP
```

When the RUN command is issued, the following occurs:

```
>RUNNH
50 NEXT I
     ^

NO MATCHING FOR
```

A program will never execute when loops are nested improperly.

Single condition branching: IF structures

There are three general formats of the IF-THEN statement pair: they direct control to a single statement, or to a series of statements (called a "subroutine"), or to an alternate statement or subroutine depending on the value of a single conditional expression, that is, whether the condition is true or false.

Branching to a single statement: The IF-THEN statement pair is frequently used to establish simple conditions for program control transfer. For example:

```
10 INPUT A
20 B=12
30 IF A<=B THEN PRINT 'A IS NOT GREATER THAN B'
40 IF A>B THEN 60
50 GOTO 80
60 PRINT 'THE VALUE OF A IS': A
65 A=A+1
70 IF A<5 THEN GOTO 10
80 END
```

The IF-THEN statement in line 30 compares the value of A to that of B. If A is less than or equal to B (which is 12), the string is printed. If A is greather than B, "THEN PRINT" is ignored. Execution then resumes with line 40. If A is greater than B, control jumps to line 60; if the condition is false, that is, if A is less than or equal to B, the GOTO in line 50 is executed, transferring control to the end of the program. If control shifts to line 60 as a result of a "true" condition in line 40, the next three lines are executed. Line 70 states yet another control condition. This time, if A is less than 5, control returns to line 10. If the condition is false, that is, if A is greater than or equal to 5, line 80 is executed and the program terminates.

Note that IF-THEN and IF-THEN GOTO are equivalent in function because the "GOTO" is assumed in the IF-THEN construct. Were the construct expressed as "IF-GOTO", the compiler would consider the omission of "THEN" to be a "fixable error".

Branching to a subroutine: Conditions can be placed on GOSUB statements by combining them with the IF-THEN statement pair. The "THEN" part of the pair is not mandatory in this construct.

```
10 INPUT A
15 IF A<=0 GOTO 90
20 GOSUB 60
25 PRINT 'RETURN PUTS US BACK HERE AT LINE 25'
30 IF A<=10 GOTO 10
40 PRINT 'THE FINAL VALUE OF A IS:':A
50 GOTO 90
60 PRINT 'BEGINNING OF SUBROUTINE'
65 B=25
70 A=(A*A) +(A*B)
80 RETURN
90 STOP
>RUNNH
!]2
BEGINNING OF SUBROUTINE
RETURN PUTS US BACK HERE AT LINE 25
THE FINAL VALUE OF A IS: 444
STOP AT LINE 90
>RUNNH
!0
STOP AT LINE 90
```

If the condition in line 15 evaluates to true, program control is transferred to the STOP statement in line 90. If false, the GOSUB statement in line 20 is activated. Control shifts to line 60, the beginning of a four-line subroutine that ends with line 80. The RETURN

statement then transfers program control to line 25, and execution resumes as directed.

Branching to alternate paths: Another form of the IF statement sets up two alternate paths for the program to follow depending on whether the indicated logical expression (condition) is true or false:

IF **expr** { THEN **stmt** / GOTO **lin-num** } ELSE { **stmt-2** / **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. For example:

```
IF I<0 THEN PRINT 'I NEGATIVE' ELSE PRINT 'I IS NOT NEGATIVE'
```

If the value of I is less than zero, the first PRINT statement is activated; if it is greater than or equal to zero, the second PRINT is executed.

This construct enables you to transfer program control or to execute some statement on the basis of a stated condition:

```
10 A =RND(X)
20 IF A<.50 THEN 40 ELSE PRINT 'A IS GREATER THAN .50'
30 IF A>.70 GOTO 45
35 X=X+1
36 GOTO 10
40 PRINT 'LINE 40:  A IS:':A
42 GOTO 50
45 PRINT 'A IS:':A
50 END
>RUNNH
LINE 40:  A IS: .2112731933594
>RUNNH
A IS GREATER THAN .50
A IS: .8529052734375
```

The first time the program is run, the "THEN 40" part of the IF statement is executed. The second RUN shows that the "ELSE" clause is executed. (The RND function in line 10 generates random numbers: see Section 10 for details.)

Branching to DO-DOEND blocks: The IF-THEN statement pair can be combined with the DO-[ELSE DO]-DOEND block to set up a multi-branched conditional structure:

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

If the logical expression, **log-expr**, evaluates to true, the statements in the DO...DOEND 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. The following program segment illustrates the use of DO-DOEND:

```
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
```

The IF statement in line 200 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. Either a THEN DO or ELSE DO statement is always used in conjunction with a DOEND statement.

Multiple condition branching

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

Branching to a statement: ON-GOTO sets up a series of line numbers to which control will be transferred depending on which condition is true.

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

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.

However, if the conditional expression is out of range, control will be transferred to the line number indicated by the ELSE GOTO line number. For example:

```
ON I GOTO 100, 200, 450 ELSE GOTO 500
```

I is evaluated and truncated to yield an integer less than or equal to the number of statement lines listed with GOTO (that is, 3). If I evaluates to 2, control shifts to line 200. If the value of I is greater than 3, control transfers to line 500. If the "ELSE GOTO" clause is omitted, an ON GOTO-GOSUB OVERRANGE error message would occur.

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

Lines 40 and 50 are included to prevent an out-of-range error should the value of I be anything but 1,2 or 3.

Branching to a subroutine: Similar to ON-GOTO, the ON-GOSUB construct transfers control to one of a series of subroutines. The subroutine chosen depends on the value of the conditional expression.

19.0

ON expr GOSUB lin-1 [,lin-2,...lin-n] [ELSE { GOTO / GOSUB } lin-n]

.
.
.
.
.

RETURN

expr, an arithmetic expression, is evaluated and truncated to an integer. Control transfer works exactly as in the ON-GOTO statement. If expr is 1, control shifts to **lin-1**, the first line of an internal subroutine. If expr is 2, control goes to **lin-2**, etc. If the conditional expression is out of

19.0 range, the ELSE GOTO or ELSE GOSUB clause takes over. Control then shifts to the line number indicated by the ELSE GOTO or ELSE GOSUB clause. 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.

The following example illustrates the use of the ON-GOSUB construct with the ELSE GOTO clause option.

```
10 INPUT X
20 ON X GOSUB 40, 70 ELSE GOTO 100
25 PRINT 'NEW VALUE FOR X IS:': X
30 IF X<5 GOTO 10
40 PRINT 'FIRST SUBROUTINE'
50 X=X+1
60 RETURN
70 PRINT 'SECOND SUBROUTINE'
80 X=X*2
90 RETURN
100 PRINT 'ELSE GOTO LINE'
110 PRINT 'X OUT OF RANGE'
120 END
>RUNNH
!1
FIRST SUBROUTINE
NEW VALUE FOR X IS: 2
!2
SECOND SUBROUTINE
NEW VALUE FOR X IS: 4
!3
ELSE GOTO LINE
X OUT OF RANGE
```

When the value of 1 is entered, the first subroutine is executed, beginning at line 40: when the value of 2 is entered, the second subroutine, beginning at line 70 is executed. When a value other than 1 or 2 is input, the ELSE GOTO clause is executed, as shown.

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 used with statement modifiers to increase control structure flexibility. Below is a list of modifiers and their respective effects on companion statements. The general format of this statement-and-modifier combination is:

statement modifier-1 condition-1 [modifier-2 condition-2]*

statement is an executable statement; **condition-1** is a logical expression; * means repeat as necessary, and **modifier** is one of the following:

Modifier	Function
IF	Execute the statements if **condition** is true.
FOR	Execute a statement or statements while **index** value meets a stated **condition.**
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.

Statements with modifiers

IF: Executable statements can be made to perform their actions only IF a specific condition is true. For example:

```
IF A<=B THEN PRINT 'A IS NOT GREATER THAN B'
IF A>B THEN 60
```

FOR: The FOR statement modifier can be used with any executable statement to establish a range of values (for an index variable) during which the statement can be executed. For example:

```
PRINT ERR$(I):I FOR I = 1 TO 30
```

This statement prints out the error message associated with the error code I for the values of I indicated. (See Section 7 for more details on ERR$.)

UNTIL: An executable statement can be made to perform some action while a certain condition remains true. When this condition becomes true, execution will terminate. For example:

```
10 Y=0
20 Y=Y+1
30 GOTO 50 UNTIL Y=10
40 GOTO 70
50 PRINT Y
60 GOTO 20
70 END
RUNNH
1
2
3
4
```

```
5
6
7
```

In this program, Y is initialized to 0. The next statement (line 20) instructs the program to increment Y by 1. Line 30 will GOTO line 50 until Y is equal to 10. When this occurs, the next line (GOTO 70) is executed, transferring control to the last statement in the program.

UNLESS: The UNLESS modifier causes a statement to be executed while a certain condition is false. Execution will continue until the condition becomes true. For example:

```
10 INPUT 'VALUE FOR A':A
20 A=SQR(A)+COS(A)
30 PRINT USING '#.###', A UNLESS A<5
35 IF A>5 GOTO 50
40 PRINT 'A LESS THAN 5'
50 STOP
```

This program prints the value of A as long as A < =5. When A is less than 5, the UNLESS condition becomes true and the PRINT statement at line 40 is executed.

WHILE: When the WHILE modifier is used, statement execution will continue as long as a certain condition remains true. The example below shows the use of the WHILE modifier:

```
10 X=8
20 FOR I=1 TO 5
25 D=X*2
30 PRINT X, X*2 WHILE D<10
40 X=X+1
50 NEXT I
60 STOP
```

The program is instructed to GOTO 50 as long as D is less than 10. The PRINT statement in line 50 is then executed. If D is greater than 10, the program is instructed to GOTO line 70.

```
10 FOR I=1 TO 5
15 X=X+1
20 D=X*2
30 GOTO 50 WHILE D<10
40 GOTO 70
50 PRINT X, X/2
60 NEXT I
70 END
>RUNNH
1                    .5
2                    1
3                    1.5
4                    2
```

Multiple modifiers: More than one modifier can be used in a single statement. Multiple modifiers are processed from right to left.

```
10 INPUT X
20 IF X>0 THEN 80 ELSE 60 UNLESS X=5
```

```
30 PRINT 'X=5'
40 PRINT 'INPUT ANOTHER VALUE FOR X:'
50 GOTO 10
60 PRINT 'X IS LESS THAN 0 '
70 GOTO 90
80 PRINT 'X IS :': X
90 END
>RUNNH
!1
X IS : 1
>RUNNH
!12
X IS : 12
>RUNNH
!5
X=5
INPUT ANOTHER VALUE FOR X:
!-10
X IS LESS THAN 0
```

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.

General format for loops with modifiers

The general format for loops with statement modifiers is:

FOR var=start [STEP incr] { WHILE / UNTIL } cond-expr

.
.
.

NEXT

Note

In FOR-loops with statement modifiers, the step size is assumed to be zero unless otherwise specified. Any value may be supplied for the STEP increment, but be careful!

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

```
10 X = 10
20 for I = 1 step 1 while I<X
30 X = X/2
40 print I, X
50 next I
60 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: The UNTIL modifier causes loop execution and variable incrementation to continue until a specified condition is met.

```
100 for I = 1 step 1 until J = 10E4
110 J = I*10
120 next I
130 end
```

On each pass through the loop, the value of J is set equal to I*10. I continues to be incremented by 1 until the value of J is equal to 10,000.

Trapping QUIT interrupts

At PRIMOS command level, hitting BREAK or CTRL-P (see Section 2), causes an immediate interruption of the ongoing operation, whatever it may be. The word "QUIT," appears at the terminal when this interrupt occurs.

In BASIC/VM, hitting CTRL-P or BREAK also causes an unscheduled halt to the ongoing operation. The response generated by the system depends on the nature of the operation and on whether or not you are at command level. To avoid unnecessary confusion of terms, the phrase, "hitting CTRL-P", will be used to describe both of these operations. This should remove any confusion between the BASIC/VM commands BREAK and QUIT, and the PRIMOS concepts just described.

In BASICV, CTRL-P's typed during execution of a command return you to BASICV command level with no displayed message. Only the " >" prompt is returned. This allows you to escape from time-consuming or otherwise undesirable operations such as LIST or TYPE.

Similarly, CTRL-P's can be used to terminate an executing program. For example, it may be necessary to get out of an infinite loop or a long PRINT operation. CTRL-P's typed during program execution return control to BASICV command level with the message:

QUIT AT LINE lin-num

lin-num is the program line at which the program was interrupted.

However, in debugging a program, it may not be convenient to keep restarting a program after each CTRL-P. An error handler, similar to the "ON ERROR GOTO" statement (see Section 7), can be used to trap each CTRL-P and resume execution at a specified point in the program.

Trapping QUITs: The ON QUIT GOTO statement can be used to redirect program control in the event of a CTRL-P occurring during program execution. Simply include an appropriate line number to which control should return. If no QUIT trap is set up, program execution will abort, and control will be surrendered to BASIC/VM command level.

It should be noted that activation of the ON QUIT GOTO handler depends on the length of the program or process from which you are attempting to escape. Because of the time-lag between actual program execution and terminal display, it may appear that execution is at a particular point when, in fact, the operation has already terminated.

Using QUIT traps: The program below sets up a QUIT trap which simply sends control to line 70 and prints the value of X at which the QUIT was effected. The actual terminal display cannot be represented on paper, so the example below has been somewhat modified for clarity. The time-delay factor makes it look as though the CTRL-P was hit when the value of X was 8: however, as the displayed message indicates, the QUIT actually occurred when the value of X was 20.

```
10 ON QUIT GOTO 70
20 FOR X=1 TO 30
```

```
30 PRINT X, X*2
40 NEXT X
50 PRINT 'DONE'
60 GOTO 80
70 PRINT 'FORCED QUIT WHERE X WAS:':X
80 END
>RUNNH
1                    2
2                    4
3                    6
4                    8
5                    10
6                    12
7                    14
8                    16
    (CTRL-P typed here)
FORCED QUIT WHERE X WAS: 20
```

If CTRL-P is typed very near the end of the loop, the ON QUIT statement is not activated, and program execution continues without interruption. The display on the screen may appear to halt temporarily, but execution still continues and the message in line 70 is not displayed.

Turning off QUIT traps: QUIT traps can be turned off with the QUIT ERROR OFF statement. The above program is modified in the example below to illustrate the difference between trapped and untrapped QUITs. After the QUIT trap has been activated, the QUIT ERROR OFF statement in line 80 turns off the trap set in line 10. The output represented here shows what is happening from the user's point of view.

The first time CTRL-P is hit, the trap is activated, transferring control to line 70. The QUIT trap is then turned off, causing the next CTRL-P to abort program execution. Control then returns to BASICV command level, with the "QUIT AT LINE 40" message display.

```
10 ON QUIT GOTO 70
20 FOR X= 1 TO 30
30 PRINT X, X*2
40 NEXT X
50 PRINT 'DONE'
60 GOTO 90
70 PRINT 'FORCED QUIT WHERE X WAS:': X
80 QUIT ERROR OFF
85 GOTO 20
90 END
>RUNNH
1                    2
2                    4
3                    6
4                    8
5                    10
   (CTRL-P typed here)
FORCED QUIT WHERE X WAS: 23
1                    2
2                    4
3                    6
```

```
4                    8
5                    10
6                    12
    (CTRL-P typed here)
QUIT AT LINE 40
```

CAUTION

Beware of trapping QUITs back to the same infinite loop from which you are trying to escape. The ON QUIT GOTO statement should not return control to a statement prior to an infinite loop you wish to exit. Such circular traps can cause your terminal to "hang". To get out of such a loop, you must log yourself out from another terminal via the LOGOUT command (PRIMOS). Simply type LOGOUT followed by your user-number. (Your user-number is assigned at LOGIN. See Section 2.) For example:

```
LOGOUT -39
```

QUIT-handling in functions: The same caution regarding the use of ON ERROR GOTO (Section 7) in user-defined functions applies to the use of ON QUIT GOTO. Be careful of making control transfers outside of the function definition. See Section 10.

BRANCHING TO EXTERNAL PROGRAMS

The COMINP and CHAIN statements direct the flow of a program to external command files or programs. They can be used either conditionally or unconditionally, depending on the context of the program in which they appear.

Transferring control to an external program

The CHAIN statement transfers program control from the currently executing program to an external (non-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 Execution of the CHAINed program begins with the first line in the program. 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 (for example, another CHAIN) is encountered.

There are two situations in which CHAIN is particularly useful:

- 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.
- A particular program may be used by several others by including a CHAIN statement in each of the calling programs.

Using CHAIN: The example in Figure 6-1 makes use of CHAIN to set up a simple "appointment calendar" system. This "system" consists of three programs, all "linked", or interrelated, by CHAIN statements. Appointment information is kept in various data files which are created as needed by these programs.

Here is a brief look at what is happening from the program control standpoint: the file CALENDAR is the "control" file from which the CHAIN sequence is begun. It begins by asking you which day of the week you want to consult. The program then asks what you want to do with the file. To simply update the information, type "UPDATE"; to display existing information, type "DISPLAY". To do both, type "BOTH". The "CHAIN A$" statement

automatically shifts program control to either the DISPLAY or UPDATE file so that the appropriate action can be performed.

The UPDATE file requests the necessary information and writes it to a data file specified by the value entered for D$. These operations make use of the file handling statements discussed in Section 8. (You don't have to know how these work to understand CHAIN.) After the requested information has been recorded, the program asks if you want to display the information just recorded in the data file. If you answer "YES" to this question, another CHAIN statement transfers control to the DISPLAY file. Otherwise, the program ends.

The DISPLAY file merely opens the desired data file and reads back the information in it.

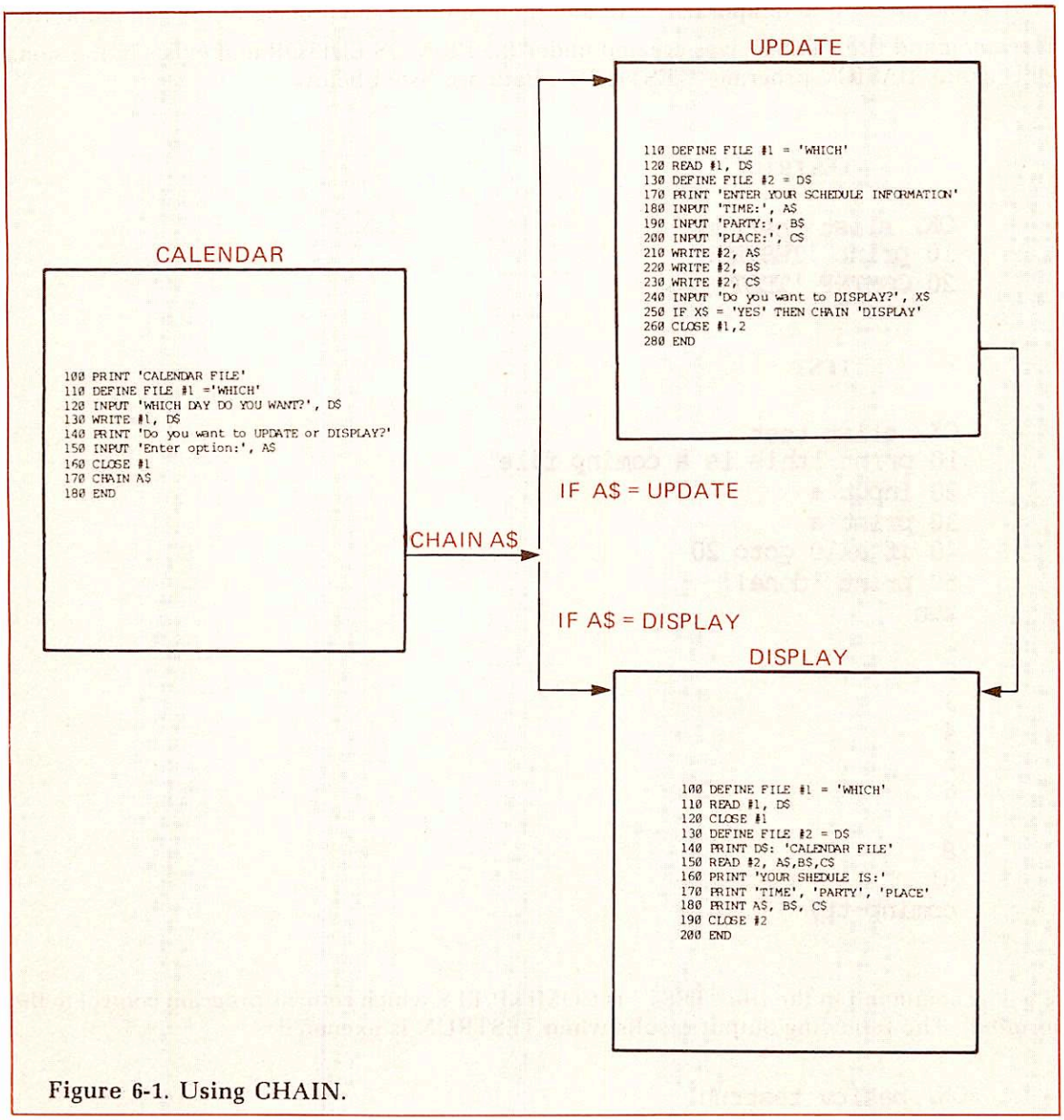


Figure 6-1. Using CHAIN.

Control transfer to command files

The COMINP statement, followed by a quoted filename argument, stops the current program flow, calls the specified external file called (command file), to the foreground, then |19.0

reads and executes the commands in it. The command file essentially takes the place of input from the terminal. This is useful when a series of commands must be frequently executed by more than one program. The series of commands and associated data can be put in a command file and then be called with COMINP from 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.

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
10 print 'TESTRUN'
20 COMINP 'TEST'
```

TEST

```
OK, slist test
10 print 'this is a cominp file'
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. The following output results when TESTRUN is executed:

```
OK, basicv testrun
TESTRUN
>10 print 'this is a cominp file'
>20 input a
>30 print a
```

```
>40 if a<10 goto 20
>50 print 'done!'
>run
TESTRUN      FRI, APR 24 1981          17:22:08


this is a cominp file
!1
1
!2
2
!3
3
!4
4
!5
5
!6
6
!7
7
!8
8
!91
91
done!
STOP AT LINE 50
OK, cominp tty
OK,
```

Note that the PRIMOS "OK" prompt is returned, indicating that it is ready to accept input from the terminal.

THE FORTRAN CALL INTERFACE

With the FORTRAN call interface users can:

- Call any shared system library routine from BASIC/VM. See the **Subroutines Reference Guide** for details on system library routines and the BASIC/VM interface.

- Call non-system library (user-written) routines loaded with BASIC/VM. In the interest of system security, only your System Administrator or supervisor should have write access – or authorize write access to others – for using the files and directories that load these routines. Appendix F and the **System Administrator's Guide** list a detailed procedure for loading non-system library routines.

Note

Only FTN routines are guaranteed to be callable, but any language's routine may be called as long as the data is interpreted correctly.

18

Declaring a Subroutine

Before any routine can be used, it must first be declared using this format:

SUB FORTRAN subr-name (arg-format [,arg-format] . . .)

The word FORTRAN is required regardless of the type of subroutine called. **subr-name** is the name of any declared legal system or non-system library routine: it can be a maximum of six characters. **arg-format** is any applicable FORTRAN declaration from the set: INTEGER, INTEGER*4, REAL or REAL*8. The word INTEGER may be abbreviated INT. The maximum number of arguments allowed is 10.

Calling the Subroutine

Once a subroutine has been declared, it can be called from within a BASIC/VM program, using this format:

CALL subr-name (arg [,arg] . . .)

A maximum of 10 arguments (**arg**) may be used.

18

Data Conversion

When an external routine is declared in BASIC/VM, the system can then recognize CALL statements and compile the appropriate code. This code consists of a dynamic link to the system routine and appropriate code for data conversion. These are contained in the code generated by BASIC/VM and will run without recompilation when a BASIC/VM binary image file is executed.

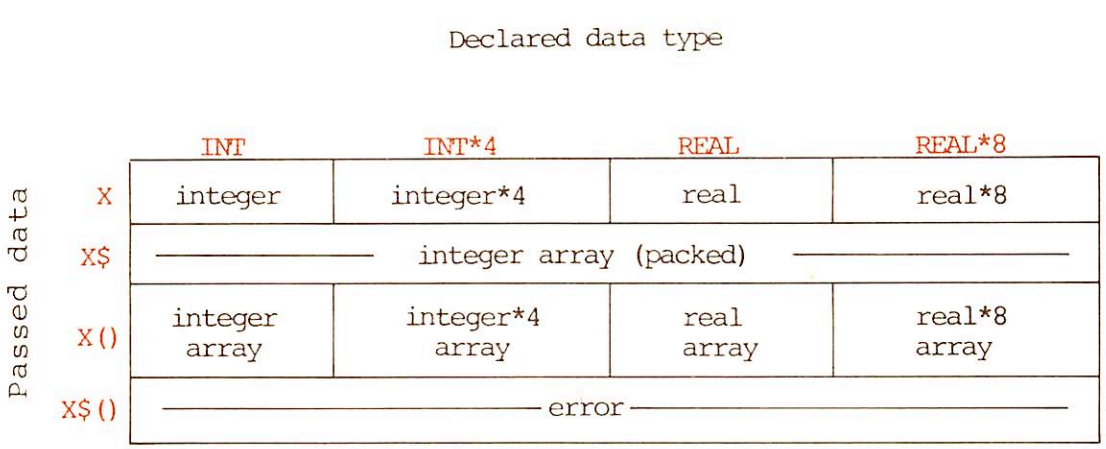
Data conversion is done automatically at call-time. The **arg-formats** given above are sufficient to convert almost any kind of data a user may encounter. Scalars and arrays are converted to their INT or REAL counterparts, if required, recalling that the BASIC/VM numeric data type is REAL*8.

Note

When an array is being passed as an argument in an external call, the zeroth element will be included. Thus, the first element of a returned array is A(0).

Call-by-reference setting is supported by this interface, but strings are restricted in that the called routine may not alter a string beyond its passed length, or unpredictable errors will result. If a string is to be returned by a called subroutine, the string variable must be pre-allocated (see the TIMDAT.BASIC example, line 25).

The following table shows how each kind of data is converted.



18

Sample Programs

The following BASIC/VM programs show how the FORTRAN CALL feature can be used in three different situations:

- The SAMPLE.TST program calls a user-written FORTRAN subroutine called SAMPLE. The SAMPLE subroutine is included in the BASICV UFD on the master disk.
- The T1 program calls a system library routine TNOU, which is written in FORTRAN.
- The TIMDAT.BASIC program calls a library routine TIMDAT which returns an array of mixed ASCII and INTEGER items.

```
SAMPLE.TST      MON, JUN 16 1980          14:39:20


10 ! THE FOLLOWING BASIC/VM PROGRAM SERVES AS A SAMPLE SESSION
20 ! TO DEMONSTRATE FTN CALL FUNCTIONALITY.
30 !
40 ! THE SECOND LINE IS OUTPUT BY THE FORTRAN SUBROUTINE 'SAMPLE'
50 ! AND 'SAMPLE' REQUIRES AN ARGUMENT OF TYPE INTEGER*2. IT RETURNS
60 ! AN ARGUMENT TWICE ITS ORIGINAL VALUE.
70 !
80 SUB FORTRAN SAMPLE (INT)
90 A = 12345
100  PRINT 'VALUE OF ARGUMENT TO BE PASSED :   ';A
110 CALL SAMPLE (A)
120 PRINT 'VALUE OF ARGUMENT RETURNED :   ';A
130 END
>RUN
SAMPLE.TST      MON, JUN 16 1980          14:39:26


VALUE OF ARGUMENT TO BE PASSED :   12345
VALUE OF ARGUMENT PASSED :      12345.00000
VALUE OF ARGUMENT RETURNED :   24690
```

18

```
T1      MON, JUN 16 1980          14:39:40


1 !
2 ! SAMPLE FTN CALL PROGRAM
3 !
4 ! NOTE THAT 'TNOU' IS A LIBRARY ROUTINE THAT PRINTS A STRING ARGUMENT
5 ! AT USER TERMINAL.
10 SUB FORTRAN TNOU(INT,INT)
20 A(I) = CODE(SUB('ABCDE',I)) FOR I = 1 TO 5
25          FOR I=1 TO 5
30              CALL TNOU(A(I),2)
35          NEXT I
40 END
>RUN
T1      MON, JUN 16 1980          14:39:44


A
B
C
D
E
```

```
TIMDAT.BASIC      MON, JUN 16 1980          14:40:02


1  !  The following program calls library routine TIMDAT which
2  !  returns an array of mixed ASCII and INTEGER format elements.
3  !  In the program, notice that TIMDAT is being called twice.
4  !  Once with array A as the return argument and the other time
5  !  with string A$ as the return argument.
6  !  Note that (1) Values returned start at A(0).
7  !            (2) Storage space MUST be allocated for A$
8  !                before the call.
9  !
10 SUB FORTRAN TIMDAT(INT,INT)! Declare FORTRAN subroutine TIMDAT
15 DIM A(15)
20 CALL TIMDAT(A(),15) !  call TIMDAT using array A as return argument
25 A$=SPA(30) ! storage space must be allocated
30            ! for A$ before the call.
35 !
40 CALL TIMDAT(A$,15) ! call TIMDAT using string A$
45 !
50 ! Note that the first three and last returned array elements are
55 ! in ASCII format. Hence no conversion is necessary when used
60 ! as strings. The other returned array elements are returned as
65 ! integers. Hence best retrieved through array A which is numeric.
70 !
75 PRINT 'MONTH :':LEFT(A$,2)   ! first returned array element
80 PRINT 'DAY :':MID(A$,3,2)   ! second returned array element
85 PRINT 'YEAR :':MID(A$,5,2)   ! third returned array element
90 PRINT 'TIME IN MINUTES SINCE MIDNIGHT :':A(3)    ! fourth
95 PRINT 'TIME IN SECONDS :':A(4)   ! fifth
100 PRINT 'TIME IN TICKS :':A(5)   ! sixth
105 PRINT 'LOGIN NAME :':right(A$,25) ! last returned element(3 words).
110 end
>RUN
TIMDAT.BASIC      MON, JUN 16 1980          14:40:12


MONTH : 06
DAY : 16
YEAR : 80
TIME IN MINUTES SINCE MIDNIGHT : 880
TIME IN SECONDS : 13
TIME IN TICKS : 319
LOGIN NAME : MHUI
```

18

7

Editing and debugging

INTRODUCTION

The BASIC/VM features covered in this section are designed to help you modify, improve, test and debug your BASIC programs. This information includes:

- Editing methods—including BASIC/VM's own line-oriented "Editor"
- Debugging procedures—including error traps and execution tracing
- Performance measurement package—measures overall program efficiency

Program errors

Errors in a program are basically of three types: syntax errors, which are violations of language rules: execution errors, which occur when a program attempts an illogical or impossible action (sometimes fatal): and logic errors, or faults in program logic, which can produce undesirable results like program failure.

Section 3 explains how the majority of program errors can be found and corrected. This section describes some additional editing commands as well as the debugging and performance measurement features which may be useful in detecting the more troublesome program errors (bugs).

EDITING A BASIC/VM PROGRAM

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

Command	Function
DELETE	Deletes one or more statements from a program.
EXTRACT	Deletes all lines except those specified.
ALTER	Edits individual lines.
RESEQUENCE	Renumbers statements after edit.
LENGTH	Determines number of lines in a file.

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** }

For example, delete line 100, lines 130 through 160 (inclusive), and line 195, in a program, type:

```
DELETE 100, 130-160, 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 individual 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. ALTER subcommands are also discussed in Section 13. Here is the complete list of subcommands:

Subcommand	Function
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 $\leq$ c, or delete until character $\geq$ 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.

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.

```
1.  > ALTER 100
    100 IF X=Y GOTO 230
    : C=E1I/>/F

    100 IF X>Y GOTO 230

    :Q
```

```
2.  >ALTER 230
    230 PRINT 'TOO LOW'
    : M11E3A/HIGH'/

    230 PRINT 'TOO HIGH'

    :Q
```

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 DEPOCITS   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 END
>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
>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. For 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 BASIC/VM 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 100, the first line of the program, and 10, respectively.

```
>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
35 END
```

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. The process of finding and fixing these errors in a program is usually called "debugging". The debugging process can be simplified through the use of the BREAK and TRACE commands, and the ON ERROR GOTO and PAUSE statements. See also **PERFORMANCE MEASUREMENT**, at the end of this section.

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 CONTINUED 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, turns off any or all previously set breakpoints. If no line numbers are specified with BREAK OFF, all breakpoints are eliminated.

The following program demonstrates how to use breakpoints 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, a message is displayed indicating the temporary halt. Type CONTINUE to resume the program after a breakpoint. For example:

```
10 PRINT 1
20 PAUSE
```

```
30 PRINT 3
40 END
>RUNNH
1
PAUSE AT LINE 20
>CONTINUE
3
```

Tracing statement execution

The TRACE ON command should be issued immediately after compiling a program, and just prior to executing it. The TRACE feature is useful for examining the path of program execution, thereby expediting the debugging process.The line number of each statement executed is displayed in brackets, for instance, [120]. 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.

```
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.

Setting error traps

There are several ways to set up error traps within a program. The ON ERROR GOTO statement establishes a line number to which control will be transferred when a run-time error occurs. For example, if invalid data is input, control can be transferred to a statementA variation of the ON ERROR statement provides for redirection of program flow if an I/O error occurs on a specified unit. 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 on this and other data file I/O.

Turning off error traps

The ERROR OFF statement cancels all error traps established by ON ERROR GOTO statements. The format is:

ERROR OFF

The example below uses ERROR OFF to cancel error trapping after line 120 has been executed.

```
10 ON ERROR GOTO 120
20 INPUT 'ENTER A VALUE FOR X:':X
30 ON X GOSUB 60, 90
40 PRINT 'X IS': X
50 GOTO 160
60 PRINT 'FIRST SUBROUTINE'
70 X=X+1
80 RETURN
90 PRINT 'SECOND SUBROUTINE'
100 X=X*2
110 RETURN
120 PRINT 'ERROR TRAP ACTIVATED'
130 PRINT 'NOW TURN TRAP OFF'
140 ERROR OFF
150 GOTO 20
160 END
>RUNNH
ENTER A VALUE FOR X:12
ERROR TRAP ACTIVATED
NOW TURN TRAP OFF
ENTER A VALUE FOR X:12
ON GOTO-GOSUB OVERRANGE ERROR  AT LINE  30
```

The ON-GOSUB statement in line 30 transfers program control to one of the two line numbers listed, provided that the value of X is either 1 or 2. When an out-of-range value is entered for X, as shown, the error trap is activated and control shifts to line 120. The ERROR OFF statement in line 140 turns off error trapping and returns control to line 20. Thereafter, if a number other than 1 or 2 is entered, the compiler prints out the error message shown above, and program execution is aborted.

Identifying locations and codes of errors

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

Function	Description
ERR	Identifies the code number of the trapped error.
ERL	Identifies the line number at which an error occurred.
ERR$(num-expr)	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:

```
10 INPUT X
15 ON ERROR GOTO 100
20 ON X GOSUB 40, 70
25 PRINT 'X IS': X
30 IF X<5 GOTO 10
35 GOTO 130
40 PRINT 'FIRST SUBROUTINE'
50 X=X+1
60 RETURN
70 PRINT 'SECOND SUBROUTINE'
80 X=X*2
90 RETURN
100 PRINT ERR$(ERR): 'AT LINE' :ERL
110 IF ERR=55 THEN DO
115 PRINT 'NO SUBROUTINE EXECUTED'
120 GOTO 160
130 DOEND
135 ELSE DO
140 PRINT 'TRY AGAIN!'
145 GOTO 10
150 DOEND
160 END
>RUNNH
!1
FIRST SUBROUTINE
X IS 2
!2
SECOND SUBROUTINE
X IS 4
!3
ON GOTO-GOSUB OVERRANGE ERROR  AT LINE 20
NO SUBROUTINE EXECUTED
>RUNNH
!12
ON GOTO-GOSUB OVERRANGE ERROR  AT LINE 20
NO SUBROUTINE EXECUTED
>RUNNH
!1
FIRST SUBROUTINE
X IS 2
!2
SECOND SUBROUTINE
```

```
X IS 4
!TEN
INPUT DATA ERROR AT LINE 1Ø
TRY AGAIN!
!76
ON GOTO-GOSUB OVERRANGE ERROR  AT LINE 2Ø
NO SUBROUTINE EXECUTED
```

In this program, an error trap condition is set in line 15. If the value input for X is out of range, that is, evaluates to anything but 1 or 2, the error trap will be activated. Control then shifts to line 100, where the special error trap functions described earlier are used to identify the nature and location of the error. The condition in line 110 then directs program control on the basis of the code of the trapped error. The complete list of error codes can be found in Appendix C. If the code (ERR) of the trapped error is 55, the first branch of the DO-DOEND set is expected. The message in line 120 is printed and execution terminates. Any other error causes the ELSE DO branch to be executed. Control then returns to the first line in the program, as shown in the final RUN of the program.

PRIMOS condition mechanism

Although most errors can be handled within the BASICV subsystem by the error-trap statements discussed earlier, there are some situations which cannot be managed by the compiler. If a "disastrous" event, such as an access violation, or a floating-point error occurs during execution of a program containing no provision for error-handling, control returns to PRIMOS, and a special message appears at the terminal. The message has the form:

```
Error:  condition "condition" raised at "address"
  [additional information]
```

A message of this form indicates that the PRIMOS error condition-handler, the "Condition Mechanism", (see Appendix D, Glossary) has been activated. PRIMOS has a default system "on-unit" which traps about 36 types of errors. It is activated only when there is no other on-unit designated to take control in the event of an error. FORTRAN, PMA and PL/I programmers can write their own on-units, or error-handlers, to rescue a program in distress. However, this feature is not needed in BASIC/VM, because a similar capability is already provided by the BASIC/VM error handling statements. Therefore, most BASICV programmers don't have to worry about condition mechanism, except to be aware of its existence.

When the system default on-unit is activated, you can do little but return to the BASIC subsystem and attempt to debug the program. If there is no error trap in the program already, it's a good idea to include one.

If the same PRIMOS error message is encountered repeatedly, for a reason not obvious to you, start a COMOUTPUT file (see Appendix D for details) and run the program again. After control returns to PRIMOS, close the COMO file, and make a paper copy of it for future reference. This file is a record of any error messages incurred during program execution, and may be useful to the system operator in hunting down possible system-level problems.

You may also want to run BASIC/VM debug (TRACE) or "Performance Measurement" tests on the program while the COMO file is open. Performance measurement, a very useful debugging tool, is discussed below.

PERFORMANCE MEASUREMENT

The performance measurement feature of BASIC/VM enables the user to:

- Measure program efficiency
- Optimize BASIC/VM code

In addition, performance measurement testing may prove a valuable asset to the debugging process.

Initiating measurement

The performance measurement feature is activated by issuing the PERF ON command prior to program compilation, that is, before a RUN or COMPILE command is issued. In order for measurements to be made during execution, special "markers" must be inserted in the code during program compilation. Thus, the PERF ON command must precede the compilation process or no measurement results will be obtained.

The program is then EXECUTEd or RUN, and the actual measurements are made. Each time the program is RUN or EXECUTEd, the old statistics are erased and replaced by new ones.

The complete PERF command format is described below.

The PERF command

The PERF command has the following format:

```
PERF { ON    }
     { OFF   }  [lin-num-1 [-lin-num-2]]
     { TABLE }  [screen-size]  [ CNT ]  [lin-num-1 [-lin-num-2]]
     { HIST  }                 [ AVG ]
                               [ TTL ]
```

The options are:

Option	Description
ON	Turns on performance measurement.
OFF	Turns off performance measurement.
TABLE	Displays performance data in tabular form.
HIST	Displays performance data in histogram form.

The TABLE option: The TABLE option displays all the statistics gathered during program execution. See **Measurement Statistics**, below.

The HIST option: The HIST option takes a number of arguments which govern the particulars of data display. Any one or all of the statistics gathered during performance measurement may be displayed in the same histogram. For a description of the AVG, CNT and TTL options, see **Measurement Statistics**, below.

screen-size is an optional numeric item which defines the width, or margin value, for the terminal screen during a histogram display. The default is the currently set margin value. The default margin value is 80 characters. **lin-num-1** specifies the program line number at which to begin displaying data. **lin-num-2** indicates the program line number at which to stop displaying performance data. (Default is last line in program.) In a histogram display, the "." character represents AVG data, the "+" character represents TTL, and "*" represents CNT.

Measurement statistics

Measurement statistics show exactly which program lines are being executed and how long each statement takes to execute. This feature is very useful in program debugging, as it allows the user to spot unexecuted program lines, as well as areas of code where performance is "hanging up."

The performance statistics gathered during program execution are identified by the following mnemonics:

Mnemonic	Description
AVG	The average time each statement took to execute.
CNT	The number of times each statement is executed.
DEV	The standard deviation of execution time.
SN	The line number of each statement executed.
SQSUM	The total squared-sum of each statement's running time.
TTL	The total running time of each statement.

All times are measured in "ticks", typically 3.03 msec. The standard deviation data can be used in assessing the data dependence or time-sharing dependence of a statement's execution time. Average statement times near one tick will usually have high standard deviations. Therefore, the standard deviation value should be taken into account when calculating average statement execution times.

Using performance measurement

The program used in this example contains a FOR-loop with an UNTIL modifier. The performance measurement data shows that the statements which make up the loop are each executed four times. The relative execution times for every statement in the program can be compared by checking the TABLE output.

```
10 X=5
20 FOR I=1 STEP 1 UNTIL Y>3
30 PRINT X/I
35 Y=Y+1
40 NEXT I
>PERF ON
>COMPILE
>EXECUTE
5
2.5
1.666666666667
1.25
STOP AT LINE 40
>PERF TABLE
    SN        CNT         AVG         DEV         TTL         SQSUM
    10          1         0.00        0.00          0             0
    20          1         0.00        0.00          0             0
    30          4         1.75        0.83          7            15
    35          4         0.25        0.43          1             1
    40          4         2.50        4.33         10           100
>PERF HIST AVG
AVERAGE  :
   10  .
   20  .
   30  ..................................................
   35  ........
   40  ............................................................
>PERF HIST TTL
TOTAL :
   10  +
   20  +
   30  ++++++++++++++++++++++++++++++++++++++++++++++++++
   35  ++++++++
   40  ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
>PERF HIST AVG CNT TTL
COUNT  AVERAGE  TOTAL :
   10  *******                .                    +
   20  *******                .                    +
   30  ***********************.................    ++++++++++++++++
   35  ***********************...                 +++
   40  ***********************.....................++++++++++++++++++++++
```

The first histog m displays the average execution times of each statement (AVG), in ticks. The second one displays TTL, the total running time of each statement, also in ticks. The third histogram displays all three options, including average statement execution times (AVG), the number of times each statement was executed (CNT), and the total running time of each statement (TTL).

Obtaining "partial" statistics

While PERF ON is in effect, each time a program is COMPILED or RUN, any existing (old) measurement data are erased. New measurements are then made. EXECUTE, unlike COMPILE or RUN, does not erase these existing measurement statistics. This allows a user to obtain "partial", or "intermediate", results, and to continue gathering performance data on a program after a BREAK or PAUSE without erasing any previously gathered data. Consequently, typing CONTINUE will allow the user to observe measurement data obtained prior to the BREAK or PAUSE.

Spotting bad code

The CNT entry of the TABLE display indicates the number of times each statement is executed. An entry of 0 (zero) in the CNT column means that the statement was never tested or executed.

Restrictions

Binary program files cannot be tested with the performance measurement feature. In fact, a binary program cannot even be COMPILEd successfully if the PERF ON command is in effect for any reason. Remember to issue PERF OFF before trying to run a binary program.

IV

ADVANCED FEATURES

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 (like 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.

ASCII sequential is the BASIC/VM default file type. Its storage and access features are suitable for many 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 routine file handling operations. 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, LIST
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
Writing a matrix to a file	MAT WRITE
Reading data into a matrix	MAT READ[*]
Closing a data file	CLOSE
Deleting a SEGDIR 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:

- Reserves buffer space in memory for data storage.
- Names a file.
- Assigns a particular file type to the file.
- Sets the record size for the new file.
- Restricts I/O operations to reading or writing.

The DEFINE statement

The general format of the DEFINE statement is:

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

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 open 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 is displayed.

Filename

Each data file opened must be assigned a name, as represented by **filename**, a legal BASIC string expression. A string variable may also be used in place of the string expression. For example:

```
10 INPUT A$
20 DEFINE FILE #3 = A$
```

The name of the file opened on unit #3 is defined by the string contained in A$.The value of A$ depends on the input received from the terminal in response to the INPUT statement in line 10.

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 60 words, or 120 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. The maximum record size is 1024 words.

Record lengths: 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 keywords READ and APPEND place restrictions on I/O operations that can be performed 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 positions 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.

Opening a SCRATCH file

A temporary, or SCRATCH, file may be opened with an abbreviated form of the DEFINE statement:

Table 8-1. File Type-Codes

Type-Code	Access Method	Contents
ASC (default)	SAM	ASCII data, formatted like terminal output with spaces as data delimiters. Commas, colons and semicolons define 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 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 10 at the start of each record. Can 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. Comma delimiters inserted.
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. Provides maximum precision and speed of access, 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	SEGDIR	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	MIDAS	Multiple Index Data Access files. Created by Prime-supplied MIDAS utilities.

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 specified. The record size can be optionally specified; the default size is 60 words. The name convention for SCRATCH files is: T$nnnn.

ACCESS METHODS

Retrieval of data from files is accomplished by one of these four methods:

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

Each access method corresponds to a particular file structure. These structures are illustrated in Figure 8-1. For a representation of MIDAS file structure, refer to the **MIDAS User's Guide** 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 the sequential and direct access features and are discussed later in this section.

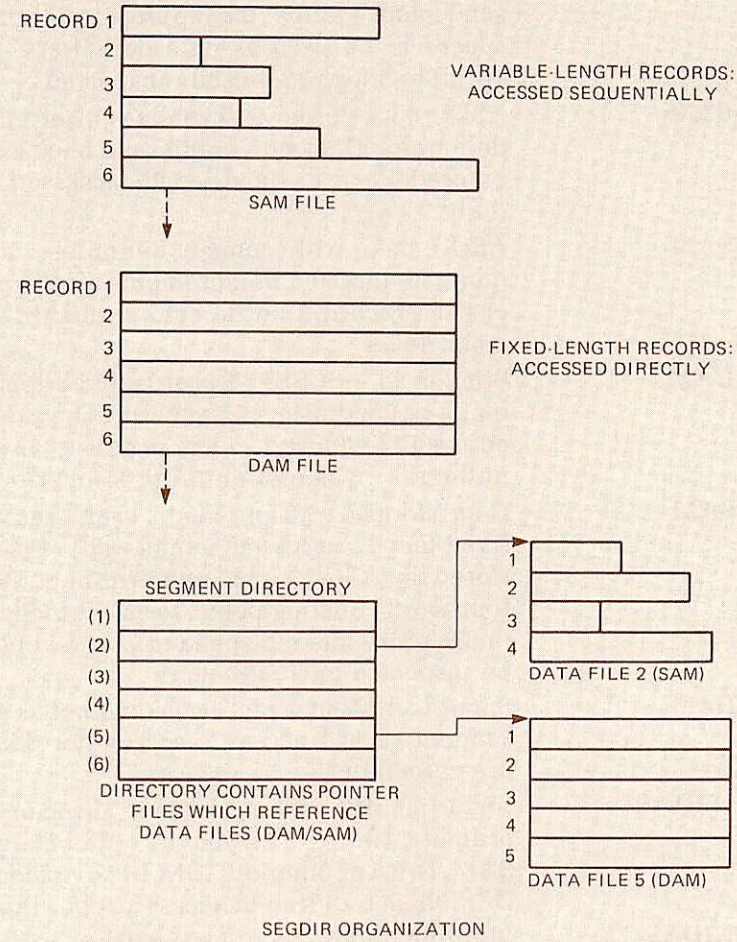


Figure 8-1. File Structures

More details on the properties of each file type can be found in Appendix E, **Advanced File Handling**. The properties of the default file type, ASC, are discussed at length and compared with the corresponding features of other ASCII file types. Users considering extensive use of file handling in BASIC/VM should first investigate the features of each file type.

SAM FILE HANDLING

Sequential files can be opened and manipulated by the following set of sequential access statements:

Statement	Function
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 appropriate type to the the opened file and advances the pointer to the next record after each WRITE.
READ [*] READLINE	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
```

Writing 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 ASCII sequential file type. By TYPEing each file (using the TYPE command), the data storage patterns of each file type (except binary) can be inspected.

```
10 DEFINE FILE #1 = 'FILE1'
20 DEFINE FILE #2 = 'FILE2', ASCSEP
30 DEFINE FILE #3 = 'FILE3', ASCLN
40 A=12
50 B$='TWELVE'
60 FOR N= 1 TO 3
70 WRITE #N, A
80 WRITE #N, B$
90 NEXT N
100 CLOSE #1,2,3
>RUNNH
STOP AT LINE 100
>!LOOK AT THE CONTENTS OF EACH FILE:
>TYPE FILE1
12
```

```
TWELVE
>TYPE FILE2
12,
TWELVE,
>TYPE FILE3
   10 12,
   20 TWELVE,
```

Writing to ASC files

Default (ASC) files have unique properties which affect the data written to them. Some of the properties are discussed below.

In ASC (default) files, 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 by forcing a different number of spaces between items in a record:

- A COMMA causes the next item to be placed in the next print zone.
- A SEMICOLON causes the next item to be placed in the next character position.
- A COLON causes the next item to be placed one character position from the current item.

The following program writes data to an ASC utilizing each type of delimiter. Each line output after the "TYPE ASCII" command represents the contents of a logical record.

```
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              2728
22 23 24 25 26 27 28
22                    23                    24 25 262728
```

Writing formatted data to a file

The WRITE USING statement is similiar to the PRINT USING statement described in Section 5. Format strings are used in WRITE USING just as they are in PRINT USING. They are composed of special characters listed in Tables 14-2 and 14-3. A summary of PRINT USING features, applicable to WRITE USING as well, can also be found in Section 14. Formatted data can be written to any type of ASCII file. However, an attempt to write formatted data to binary file will generate an error message.

WRITE USING format: 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, represented by **item-1 -item-n**, to be formatted. For example:

```
10 DEFINE FILE #1 = 'EXAMPLE'
20 WRITE #1 USING '$###.##', 120
30 WRITE #1 USING '<#####', 'FUNNY'
40 WRITE #1 USING '>#####', 'FUNNY'
50 WRITE #1, 120
60 WRITE #1, 'FUNNY'
70 CLOSE #1
>RUNNH
>TYPE EXAMPLE
$120.00
FUNNY
 FUNNY
120
FUNNY
```

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

```
90 DEFINE FILE #1 = 'FILE'
100 WRITE #1 USING '$###.##', 12000
110 REWIND #1
120 READ #1, A$
130 PRINT A$
>RUNNH
*******
```

The second WRITE USING statement in line 30 left-justifies a string datum in the file with the left-angle bracket (<) symbol. The WRITE statement in line 40 writes the same item with right-justification. The data written to a file with WRITE USING is stored exactly as specified by the format string.

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 prior to the current record, use the REWIND statement to reposition the pointer to the top of the file. The READ pointer cannot be positioned to random records 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 positions to the top of the file unless otherwise instructed, as with the "APPEND" option of the DEFINE command. For example:

```
10 DEFINE FILE #1 = 'ASC'
20 WRITE #1, 12
30 READ #1, A
>RUNNH
READ AFTER WRITE ON SEQUENTIAL FILE  AT LINE  30
```

To READ the data in the file, the program must be modified, as in:

```
10 define file #1 = 'ASC'
20 write #1, 12
30 rewind #1
40 read #1, A
50 print A
60 close #1
70 end
>runnh
12
```

The READ* statement

A variation of the READ statement, READ*, holds the file pointer 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 to be performed on this record. The details of READ* are illustrated in the example below. There are a few points to keep in mind:

- If a default READ (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 in the current READ list.

```
10 DEFINE FILE #1 = 'READ.FILE', ASCLN
15 WRITE #1, 100, 200, 300
20 WRITE #1, 400,500,600
25 REWIND #1
30 PRINT 'FIRST READ WITHOUT *'
35 READ #1, A
40 PRINT A
45 PRINT
50 READ #1, B
55 PRINT B
60 PRINT
```

```
65 REWIND #1
70 PRINT 'NOW READ WITH READ*'
75 READ* #1,A
80 PRINT A
85 READ* #1, B
90 PRINT
95 PRINT B
100 READ* #1, C
105 PRINT
110 PRINT C
115 CLOSE #1
120 END
>RUNNH
FIRST READ WITHOUT *
100

400

NOW READ WITH READ*
100

200

300
```

The READLINE statement

READLINE allows the contents of an entire ASCII file record, including commas, colons, semicolons and spaces, to be read as one data item. This version of the READ statement does NOT work on binary files. READLINE is especially useful when reading default ASCII or ASCSEP files which contain data with internal commas. Unlike the READ statement, READLINE does not interpret commas as data delimiters. Thus strings containing commas will not be broken up by READLINE. For example, if a record in an ASCII file opened on unit #1 contains the following:

```
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 ASCII sequential file types.

```
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 'WHIMSEY, PETER:', 'OXFORD, BALLIOL:', 'DETECTIVE, ETC.'
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

WHIMSEY

READLINE FOR FILE ON UNIT # 1

WHIMSEY, PETER:      OXFORD, BALLIOL:      DETECTIVE, ETC.

READ FOR FILE ON UNIT # 2

WHIMSEY

READLINE FOR FILE ON UNIT # 2

WHIMSEY, PETER:,OXFORD, BALLIOL:,DETECTIVE, ETC.,

READ FOR FILE ON UNIT # 3

WHIMSEY

READLINE FOR FILE ON UNIT # 3

 WHIMSEY, PETER:,OXFORD, BALLIOL:,DETECTIVE, ETC.,

READ FOR FILE ON UNIT # 4

WHIMSEY, PETER:

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 (that is, the last data record) during a READ, an END OF FILE error message is generated. This error abruptly terminates program execution. To avoid this, include an "ON END #unit GOTO" statement to tell the program what to do in the event of an END OF FILE condition. 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

All types of I/O errors that occur during data file access can be trapped with the "ON ERROR #unit GOTO" statement, a variation of "ON ERROR GOTO", discussed in Section 7. This statement is generally placed near the beginning of the program. It can be used in both SAM and DAM files, and is especially helpful when doing multiple READs from a binary file whose contents are not easily monitored.

In the absence of an error trap statement of some sort, any I/O error, including an end-of-file, that occurs during file access generates an error message and halts program execution. Inclusion of an ON END or ON ERROR statement at the beginning of the program eliminates I/O error-generated program halts, either by transferring control to another line in the program, or by performing some other action. For example:

```
10 DEFINE FILE #1 = 'END1', ASCSEP
20 ON ERROR #1 GOTO 80
30 WRITE #1, 'GOVERNOR AS OF 1979'
40 WRITE #1, 'CALIFORNIA','NEW YORK'
45 WRITE #1, 'JERRY BROWN', 'HUGH CAREY'
50 REWIND #1
60 READ #1, A$
70 READ #1, B
80 PRINT 'ERROR DURING FILE READ'
90 PRINT ERR$(ERR)
100 REWIND #1
110 READ #1, A$,B$,C$,D$,E$
120 PRINT
130 PRINT A$
135 PRINT
140 PRINT B$, C$
145 PRINT D$,E$
150 CLOSE #1
160 END
>RUNNH
ERROR DURING FILE READ
INPUT DATA ERROR

GOVERNOR AS OF 1979
```

```
CALIFORNIA              NEW YORK
JERRY BROWN             HUGH CAREY
>!REMOVE ON ERROR STATEMENT IN LINE 2Ø;
>2Ø
>RUNNH
INPUT DATA ERROR AT LINE  7Ø
```

Without the ON ERROR statement in line 20, the INPUT DATA ERROR, generated by the variable-data mismatch in line 70, the program fails.

Closing 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 which was opened for other use. If the file is not CLOSEd, it may be lost or truncated. A single CLOSE statement can close one or many file units which have been opened. For example:

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

DAM FILE HANDLING

The DAM file handling statements are identical to those used in SAM file handling, with the important exception of POSITION. Below is a list of all available DAM file handling statements.

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[*], READLINE	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 DAM file can be set to a value larger or smaller than the default of 60 words (120 characters) when the file is first defined. The minimum is 4 words; the maximum 1024. The record size of a DAM file is adjusted according to the value 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 DAM 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.

This statement defines 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 is stored in ASCDA files just as in ASCSEP files, that is 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,
```

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. The record at the top of the file is record 0, and is the first record in the file. It is positioned to with the statement:

```
POSITION #unit TO 1
```

Notice that the statement does not position to 0. The second record in the file is reached by stating:

```
POSITION #unit TO 2
```

and so forth. An out-of-range record number causes the END OF FILE message to be displayed. The data in the currently positioned record can then be obtained with a READ statement.

The LIN# function in DAM files: The LIN#(unit) function can be used in DAM file access to check the record number to which the pointer is positioned. Instead of returning a line number as it does in ASCLN files, it returns the number of the current record in the file. For example, if the statement, POSITION #1 TO 1, is issued, LIN#(unit) would return the record number as 0, not 1. The first record in the file is record 0, not record 1, according to the LIN # function.

The following example shows random access to a direct access file called "BOOKS". The file contents are displayed by using the TYPE command. The file is then re-opened for READ access only and data are retrieved with the POSITION and READ statements. The LIN# function shows where the pointer is at various stages during record access. Note that after each READ, the pointer automatically advances to the next record.

```
>TYPE BOOKS
BIBLIOGRAPHY, RE. DOROTHY L. SAYERS,
TITLE: MURDER MUST ADVERTISE,
PUB. DATE: JUNE, 1933,
SUBJECT: LORD PETER, ALIAS DEATH BREDON, WORKS FOR AD FIRM,
INVESTIGATES DEMISE OF PREDECESSOR,
>!Define a READ only file.
>DEFINE READ FILE #1='BOOKS', ASCDA, 30
>! Find out what record we're at in file.
>PRINT LIN #(1)
0
>READ #1, A$
>PRINT A$
BIBLIOGRAPHY
>PRINT LIN #(1)
1
>POSITION #1 TO 5
>READLINE #1, B$
>PRINT B$
INVESTIGATES DEMISE OF PREDECESSOR,
>POSITION #1 TO 3
>PRINT LIN #(1)
2
>READLINE #1, A$
>PRINT A$
PUB. DATE: JUNE, 1933,
```

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 is returned. In this example, the record size is set to 30 words (60 characters), as originally defined.

In direct access, the file pointer scans through a list of pointers for each record in the file and locates the desired record. The pointer determines the actual location of this record by counting the number of characters it has to bypass in order to reach the desired record. The LIN# function indicates where the pointer is positioned at any given time. For example, after an optionless READ is done, the pointer advances to the next sequential record, as shown.

Reading DAM files

DAM file READs are done in the same manner as are SAM file READs, that is, READ, READ* and READLINE work in direct access just as they do in sequential access. After each READ is completed, the file pointer is advanced to the next sequential record. READ* holds the pointer at the current record until all the variables specified in the next READ statement have been satisfied, or until the last value in the current record has been read. In ASCDA files, READLINE returns all the values in a record as one datum, commas, semicolons, 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.

Trapping error 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 direct access files as well as sequential files.

Closing 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 you are finished using them. If you are accidentally returned to PRIMOS command level during program execution due to an access violation, for example, any file units left open can be closed with the CLOSE ALL command. See 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 file type supported by BASIC/VM.

Note

Use of segment directories via the SEGDIR file type is discouraged. To obtain similar results use MIDAS. For complete details see the **MIDAS User's Guide.**

18

Creating a segment directory

Segment directories themselves contain no data in immediately accessible form. Instead, they maintain a list of pointers to individual data files which contain accessible data.

To create a segment directory, you must first set up the "template", or, "skeleton", for the directory. Unlike the other file types discussed previously, data cannot be written directly to this segment directory. Individual data files must be opened under the segment directory. The only entry in the segment directory is a list of pointers to the data files that have been created under it.

Opening a data file under a SEGDIR: To open a data file under a segment directory, you must first open, or DEFINE, the data file on any file unit not currently in use. The file unit on which the segment directory was opened should remain open throughout the data file creation process. Any type of data file described in this section can be defined under a segment directory.

Identifying a SEGDIR data file: Data files are not named in the usual manner; in fact, they are not "named" at all. Instead, they are identified by number according to the order in which they are listed under a particular segment directory.

A special "shorthand" convention is used to identify the data file as a segment directory entry. The "(SD#unit)" convention, where #unit represents the file unit number on which the segment directory is open, tells the file-handler that the file unit being opened is reserved for a data file listed under a currently open segment directory. The letters "SD" identify the previously opened segment directory. Thus, if a segment directory is opened on file unit #1, any data file created under it will be identified by the convention: "(SD1)".

Adding data to a SEGDIR file: After opening a data file, data can be added to it with the WRITE #unit statement. When you are finished working with a data file, CLOSE the file unit on which it is open.

Creating successive data files: If you want to create another data file under a particular SEGDIR, POSITION the file pointer to the next sequential record in the segment directory. Then, repeat the DEFINE, WRITE, and CLOSE sequence just described.

Each time you move the pointer and DEFINE a new file, a new entry is added to the directory, identifying the location, or address, of the new data file. If the pointer is NOT positioned to another record location, the next DEFINE statement will overwrite the data that exists in that segment location.

Setting up a sample SEGDIR

This sample program sets up a segment directory using the procedure described above. This particular program first opens a segment directory on unit #1 and calls it "SEGA". Then, two separate data files are created under it. The first file is an ASCLN file: the second, ASCSEP. As many files as desired can be added to a segment directory by following the procedure outlined above and illustrated here:

```
10 DEFINE FILE #1 = 'SEGA', SEGDIR
20 DEFINE FILE #2 = '(SD1)', ASCLN
30 WRITE #2, 'FIRST DATA FILE'
40 CLOSE #2
50 !NOW POSITION #1 TO 2 AND CONTINUE AS ABOVE
60 POSITION #1 TO 2
70 DEFINE FILE #2='(SD1)', ASCSEP
80 WRITE #2, 'SECOND DATA FILE'
90 CLOSE #2
95 !Repeat above sequence until all data files are created.
100 CLOSE #1
110 END
```

Accessing a segment directory data file

The process of opening an existing SEGDIR data file is identical to that just shown. Using the above SEGDIR as an example, the steps taken are:

1. DEFINE (open) SEGA on an available file unit.
2. Position the file pointer to the desired record entry.
3. Open the desired data file on an available file unit.
4. Use the READ or WRITE statements to perform desired file I/O.

A typical data file access situation is represented in the sample terminal session below. Tracing through the above procedure steps, the program first opens SEGA on file unit 1 with the statement:

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

The segment directory is now open on file unit 1. Next, the file READ pointer is positioned to the first entry in the directory:

```
POSITION #1 TO 1
```

The file pointer is now at segment number 1 which contains the address of data file 1.

The next step is to open this data file on a file unit, in this case, file unit #2:

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

Note that the "(SD1)" naming convention is required to identify the location of the data file. Specification of a data file type is optional. The file unit on which the segment directory is open must remain open as long as any data files under it are being accessed.

Sample data file access: The program below opens the previously defined segment directory, SEGA, on file unit #1. The POSITION statement is then used to position the file read pointer to the first record location in the segment directory. The first entry in this directory references the ASCLN data file created in the previous program. After access to the first data file is completed, the unit, #2, is closed. The POSITION statement is then used to advance the pointer to another entry and the sequence is repeated:

```
100 DEFINE FILE #1 = 'SEGA', SEGDIR
110 POSITION #1 TO 1
120 DEFINE FILE #2 = '(SD1)'
130 READ #2, A$
140 PRINT A$
```

```
150 CLOSE #2
160 POSITION #1 TO 2
170 DEFINE FILE #2 = '(SD1)'
180 READ #2, A$
190 PRINT A$
200 CLOSE #2
210 PRINT 'THAT IS THE END OF THE SEGDIR'
220 CLOSE #1
230 PRINT 'BYE!'
240 END
>RUNNH
10 FIRST DATA FILE

SECOND DATA FILE
THAT IS THE END OF THE SEGDIR
BYE!
```

Restrictions: There are several points to keep in mind when dealing with segment directories. Only one SEGDIR data file can be opened at a time, because of the "(SD#unit)" naming convention restriction. It is also illegal to attempt a READ or WRITE operation on a segment directory. For example, if SEGA is opened on unit #1, "READ #1, A$", would produce the error message:

```
ILLEGAL OPERATION ON SEGDIR
```

Nesting segment directories

Any type of file may be placed under a segment directory, including another segment directory. The process of "nesting" segment directories is exemplified below.

First, segment directory SEGB is opened on unit #1. Then another segment directory, identified by (SD1), is defined under it. Finally, keeping both file units open, a data file is defined under the "nested" segment directory. This data file is identified by the convention "(SD2)", which indicates that it is listed under the second SEGDIR which is open on unit 2.

```
10 DEFINE FILE #1 = 'SEGB', SEGDIR
20 POSITION #1 TO 1
30 DEFINE FILE #2 = '(SD1)', SEGDIR
40 DEFINE FILE #3 = '(SD2)', ASCLN
50 WRITE #3, 'DATA FILE UNDER SEGDIR WHICH IS UNDER SEGB'
60 CLOSE #3
70 CLOSE #2
80 CLOSE #1
>RUNNH
STOP AT LINE 80
```

These nested directories and data files are accessed as shown here:

```
10 DEFINE FILE #1 = 'SEGB', SEGDIR
20 POSITION #1 TO 1
30 DEFINE FILE #2 = '(SD1)', SEGDIR
40 DEFINE FILE #3 = '(SD2)'
50 READ #3, A$
60 PRINT A$
70 CLOSE #1,2,3
```

```
80 PRINT 'ALL DONE'
90 END
>RUNNH
10 DATA FILE UNDER SEGDIR WHICH IS UNDER SEGB
ALL DONE
```

Notice that both file units on which the segment directories were opened must be kept open during access to the data file under the "nested" SEGDIR. Be careful of nesting segment directories too deeply or you may run out of file units in the process!

Deleting segment directory data files

Data files in a segment directory are deleted with the REPLACE statement:

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

The parameters, x and y , are numeric items which represent pointers to two data files, (x) and (y) respectively. These files must exist under the segment directory opened on the indicated file unit. REPLACE deletes data file (x), and moves pointer y into pointer x, leaving the location at pointer y empty. The original (x) is gone, and the original (y) has been renamed (x). The program below illustrates this process. Refer also to Figure 8-2 for a visual interpretation.

The first program listed here reads the contents of several data files under the segment directory SEGA. Clearly, the three files are listed in consecutive numerical order in the directory.

```
10 DEFINE FILE #1 = 'SEGA', SEGDIR
20 POSITION #1 TO 1
30 DEFINE FILE #2 = '(SD1)'
40 READ #2, A$
50 PRINT A$
60 CLOSE #2
70 POSITION #1 TO 2
80 DEFINE FILE #2 = '(SD1)'
90 READ #2, A$
100 PRINT A$
110 CLOSE #2
120 POSITION #1 TO 3
130 DEFINE FILE #2 = '(SD1)'
140 READ #2, A$
150 PRINT A$
160 CLOSE #2
170 PRINT 'ALL DONE'
180 END
>RUNNH
10 FIRST DATA FILE
SECOND DATA FILE
THIRD DATA FILE
ALL DONE
```

The second program shows what happens when the pointer in segment 3 is moved to segment 1. Data file 1 is erased and data file 3 takes its place.

```
5 ON ERROR GOTO 220
10 DEFINE FILE #1 = 'SEGA', SEGDIR
```

```
20 REPLACE #1 SEG 1 BY SEG 3
30 POSITION #1 TO 1
40 PRINT 'FIRST ENTRY IN SEGDIR:'
50 DEFINE FILE #2 = '(SD1)'
60 READ #2, A$
70 PRINT A$
80 CLOSE #2
90 PRINT 'SECOND ENTRY IN SEGDIR:'
100 POSITION #1 TO 2
110 DEFINE FILE #2 = '(SD1)'
120 READ #2, A$
130 PRINT A$
140 CLOSE #2
150 POSITION #1 TO 3
160 PRINT 'THIRD ENTRY IN SEGDIR'
170 DEFINE FILE #2 = '(SD1)'
180 READ #2, A$
190 PRINT A$
200 CLOSE #2
210 GOTO 230
220 PRINT 'NOTHING THERE ANYMORE!!!'
230 CLOSE #1
240 END
>RUNNH
FIRST ENTRY IN SEGDIR:
THIRD DATA FILE
SECOND ENTRY IN SEGDIR:
SECOND DATA FILE
THIRD ENTRY IN SEGDIR
NOTHING THERE ANYMORE!!!
```

The REPLACE statement in line 20 deletes the first data file under the segment directory and replaces it with data subfile 3, leaving the third segment empty. The READs show what actually takes place in the segment directory when a data file is deleted. The ON ERROR statement is included to trap the READ error that occurs when a READ is attempted on the no-longer-existent data file in segment 3.

Deleting segment directories

The simplest way to delete an entire segment directory is to use the FUTIL command of PRIMOS. See Appendix D for details on FUTIL and its subcommands. Like all PRIMOS commands, it is issued from PRIMOS, not BASICV, command level. The "TREDEL" subcommand of FUTIL deletes any kind of segment directory, including MIDAS files. TREDEL is issued in response to FUTIL's right-angle bracket prompt character (>).

This example shows how a segment directory (SEGA) is deleted from PRIMOS command level. Notice that the "FUTIL" command, which can be typed in upper- or lower-case letters, is issued in response to the "OK," prompt.

```
OK, FUTIL
[FUTIL rev 17.0]
>TREDEL SEGA
> Q

OK,
```

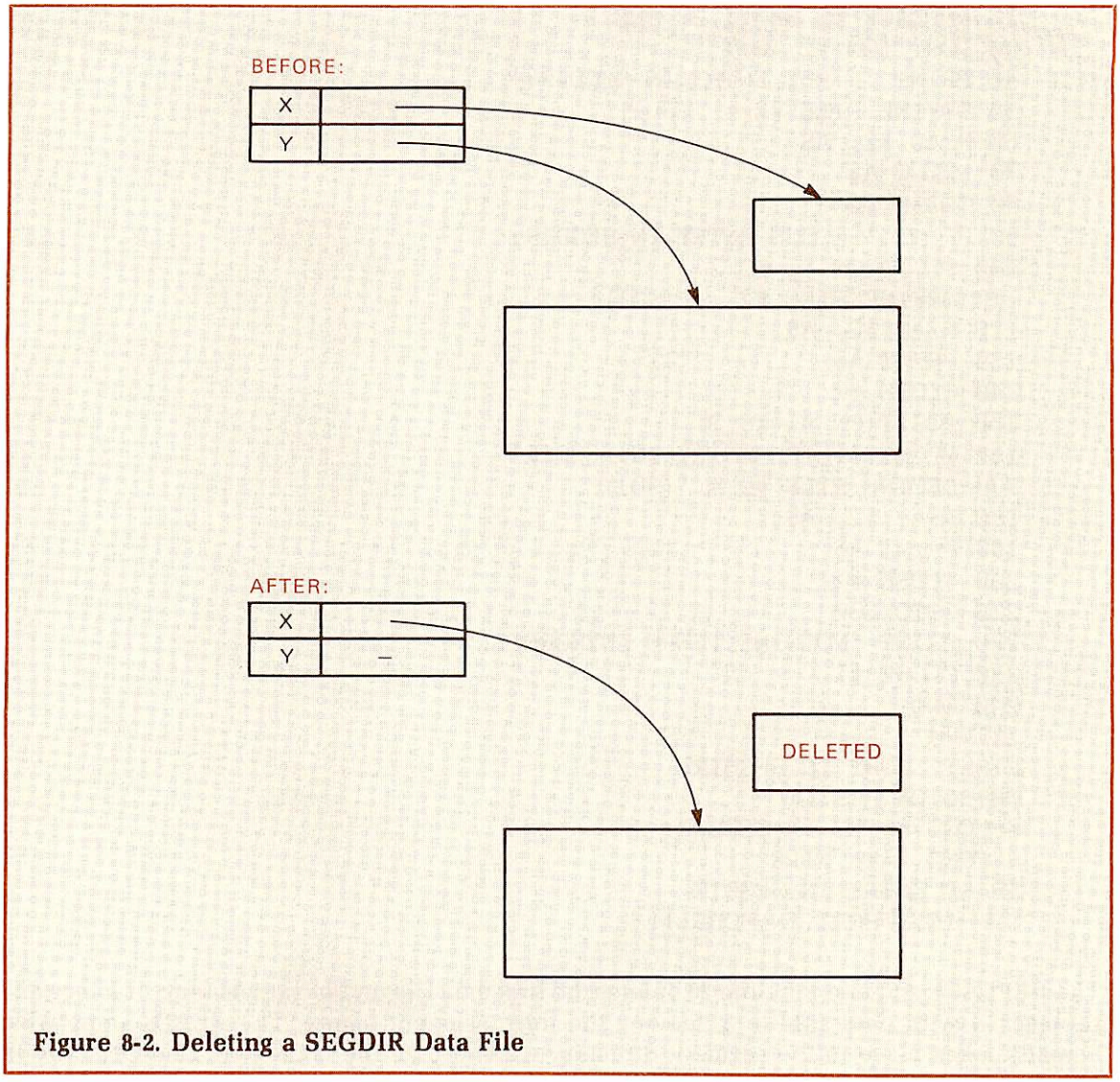


Figure 8-2. Deleting a SEGDIR Data File

MIDAS FILE HANDLING IN BASIC/VM

MIDAS, or the Multiple Index Data Access System, is a collection of interactive utilities and subroutines for managing index-sequential and direct access data file. 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
18 | access and secondary-index data. For more information on the features of MIDAS and its related utilities, consult the **MIDAS User's Guide.**

Note

18 | For an up-to-date description of all the BASIC/VM MIDAS access statements, see Section 8 of the **MIDAS User's Guide.**

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 utility CREATK, is used to do this. CREATK is invoked from PRIMOS command level; it sets up a MIDAS file template that can then be accessed by a variety of methods, including other MIDAS utilities, BASIC/VM programs and COBOL programs, to name a few.

CREATK prompts the user for input describing the file to be created. The parameters supplied include the filename, access type, and data subfile information including key type and key size for both primary and secondary indexes.

A MIDAS file can contain up to 18 indexes, that is, 1 primary and 17 secondary indexes.

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.

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 the data records is accomplished by specifying the correct "coordinates" of a particular element or key in the matrix. See Figure 8-3.

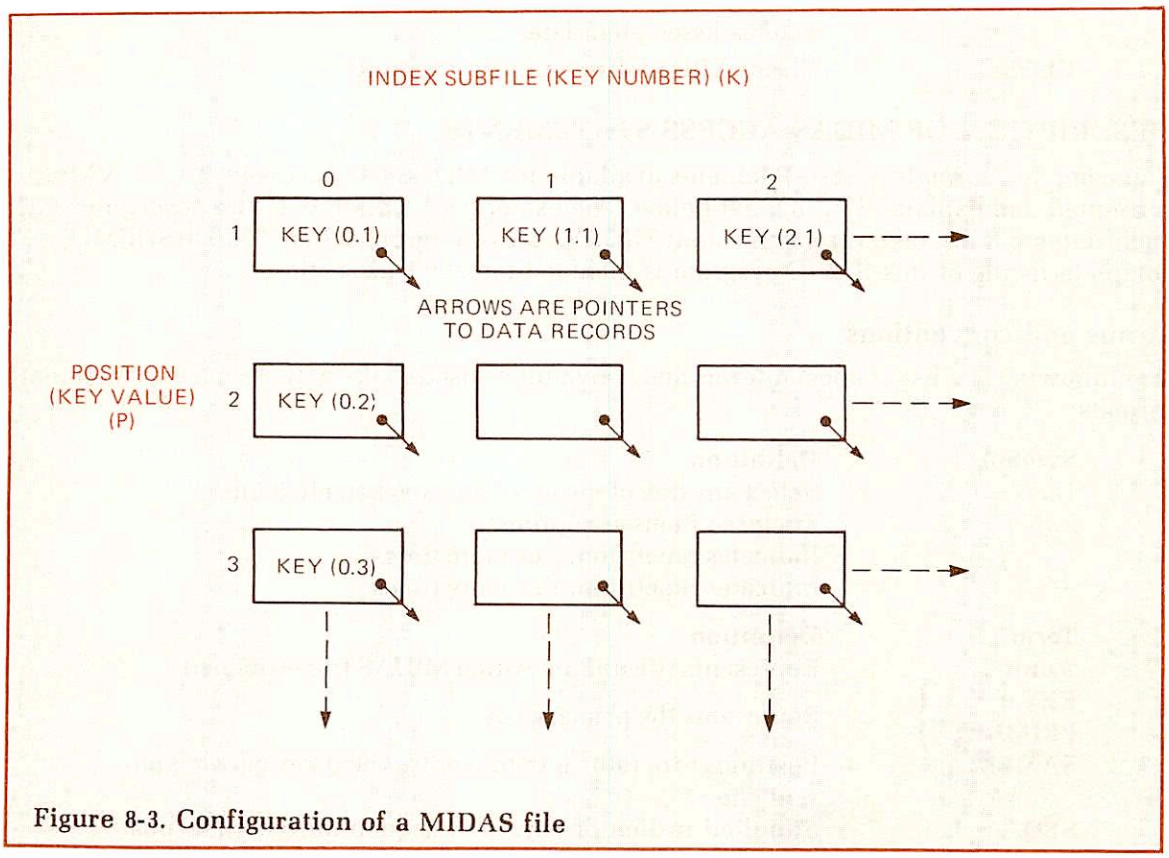


Figure 8-3. Configuration of a MIDAS file

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.

MIDAS access statements

The statements used to access MIDAS files in BASIC/VM are similar in function and format to the other file handling statements 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. These statements are listed in the table 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.

DESCRIPTION OF MIDAS ACCESS STATEMENTS

The complete formats of the statements available for MIDAS file access in BASIC/VM are presented and explained in the text below. The examples supplied with the descriptions of each statement are taken from an actual MIDAS access program called, "MIDASDEMO". A complete listing of this BASIC program is included later in this section.

Terms and conventions

The following is a list of special terms and conventions used in the MIDAS access statement formats:

Symbol	Definition
{...}	Select any one of the vertically stacked elements.
[...]	Enclosed items are optional.
*	Indicates repetition, 0 or more times.
+	Indicates repetition, 1 or more times.

Term	Definition
#unit	Represents file unit on which MIDAS file is opened.
KEY 0 PRIMKEY	Represents the primary key.
SAMEKEY	Positions to or returns datum only if next key matches current one.
SEQ	Supplied in lieu of key: next sequential record is positioned to and read.
num-expr-x	Represents numeric expression.
str-expr-x	Represents string expression.

ACCESSING MIDAS FILES

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-DEMO program following the presentation of MIDAS access 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 default record size is 60 words (120 characters). 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. The DEFINE statement format is:

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

Parameter	Definition
#unit	file unit number on which file is opened
str-expr	MIDAS filename
num-expr-1	record size (in words) (default=60)

An example of the DEFINE statement is:

```
DEFINE FILE #1= 'DIR', MIDAS, 50
```

This statement opens a MIDAS file called "DIR" on unit 1, with a record size of 50 words.

Positioning the file pointer

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

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

Parameter	Definition
num-expr	secondary key number (index subfile number)
str-expr	key value(primary or secondary)

For example:

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

Reading a MIDAS file

Data are retrieved from a MIDAS file with 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.

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

Parameter	Definition
num-expr	Index subfile number
str-expr	Key value
str-var	Variable into which data is read from record

For example:

```
READ #1, SEQ, A$
```

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, (for instance, 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.

ADD #unit, str-expr-1, { PRIMKEY / KEY [0-expr] } = str-expr-2 keylist

where keylist = [,KEY num-expr-1 = str-expr-3] *

Parameter	Definition
0-expr	Expression evaluating to zero
str-expr-1	Value of new record
str-expr-2	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

For example:

```
ADD #1, X$, KEY0 = I$(1)
```

Updating a record

The UPDATE statement overwrites the current record with a user-specified string which represents the "new" record value. 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 cannot determine changes made to a key field.

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

X$ represents the string expression which will update the current record by overwriting it.

Deleting MIDAS file keys

The REMOVE statement deletes a given key from the MIDAS data base. If the key is a primary key (where num-expr=0) 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 0 is assumed. This positions the pointer to the upper left corner of the matrix (equivalent to REWIND #unit, KEY 0).

REWIND #unit [, KEY num-expr]

For example, the following statement repositions the pointer to the top of index subfile 3:

```
REWIND #1, 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

Deleting a MIDAS file

MIDAS files cannot be deleted from within the BASICV subsystem. They can be deleted from PRIMOS command level with the FUTIL command. Use the TREDEL subcommand, as shown in the example under **Deleting a Segment Directory**, earlier in this section. See Appendix D for additional information on FUTIL.

Setting up a MIDAS file with CREATK

The first step in using a MIDAS file is to set up a template for it with the CREATK utility. The template is a skeleton for the file. It defines the keys on which data will be searched during data retrieval. CREATK prompts the user for template information as shown in the example below. The template created by this example is used by the MIDASDEMO program.

```
OK, creatk
[CREATK rev 18.0]

MINIMUM OPTIONS? yes

FILE NAME? dir
NEW FILE? yes
DIRECT ACCESS? no

DATA SUBFILE QUESTIONS

PRIMARY KEY TYPE: ascii
PRIMARY KEY SIZE = : w 16
DATA SIZE = : 64

SECONDARY INDEX

INDEX NO.? 1

DUPLICATE KEYS PERMITTED? yes

KEY TYPE: ascii
KEY SIZE = : w 16
SECONDARY DATA SIZE = :    (CR)

INDEX NO.? 2

DUPLICATE KEYS PERMITTED? yes

KEY TYPE: ascii
KEY SIZE = : w 16
SECONDARY DATA SIZE = :   (CR)
```

18

```
INDEX NO.? 3

DUPLICATE KEYS PERMITTED? yes

KEY TYPE: ascii
KEY SIZE = : w 16
SECONDARY DATA SIZE = :    (CR)

INDEX NO.?    (CR)

SETTING FILE LOCK TO N READERS AND N WRITERS
OK,
```

After the template has been set up, the record information can be added to the file with the MIDASDEMO program, listed below.

Accessing the MIDAS file

The following program, MIDASDEMO, illustrates the use of the BASIC/VM MIDAS access statements to add data to and retrieve record information from the MIDAS file set up earlier with CREATK. 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 BASICVs
8 !   string functions to automatically control string lengths,
9 !   to perform space-padding, and facilitate string comparisons.
10 !
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
```

Description of MIDAS demo program

The MIDASDEMO program sets up a series of functions to make record access more flexible. Most of the MIDAS access statements described earlier are included in the Demo and are described in the context of the program. The user-defined functions in this program make use of the string system functions listed in Table 10-2. These user-defined functions are:

Function	Description
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 these system functions: CVT$$- which reformats a given string according to the indicated mask (listed in Table 10-3); INDEX(X$,Y$)- computes the starting position of Y$ in X$. (In this case, finds first blank space in X$.) LEFT(X$,Y)- returns the leftmost Y characters of string X$.(In this case, returns first characters immediately to the left of the first blank.) RIGHT(X$,Y) - returns rightmost Y characters of string X$. In this case, returns those beginning after the first blank found in X$.

Opening the MIDAS file: After defining the functions that accept and organize input for the data file, the program opens the MIDAS file called "DIR"on file unit #1. A record size of 64 words is specified, meaning that data in excess of 64 words will not fit into a single record.

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

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

Record Access: The program then defines what will occur when the user inputs the words, "FIND" or "FIND ALL". The FIND ALL function incorporates the BASIC/VM 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 accommodates the use of duplicate keys (that is, having more than one record or entry referenced by a single key).

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

Reading record contents: 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$.

```
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.

Automatic record positioning: The POSITION statement 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 340.

Adding a new record: If the user types ADD at line 340, the program jumps to line 470 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".)

Listing all file records: 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 returned to the beginning or (upper left corner of the matrix) of the file:

```
REWIND #1
```

No key is specified, therefore, KEY 0 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
```

Duplicate keys: 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-then-read" routine is done until the end of the file is reached.

Error handling: The remainder of the program handles errors, using the BASIC/VM error-handler functions, ERR$ and ERR. See Sections 7 and 14 for details.

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 is established by the DEMO program as the input prompt.

In the sample terminal session below, 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 0), and secondary keys, KEY 1-3, respectively. The various program-defined functions discussed earlier are then utilized.

The CTRL-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.

```
NEW OR OLD: OLD DEMO
>RUNNH
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
.(CTRL-C typed here)
END OF DATA AT LINE 240
>QUIT
```

9

Arrays and matrices

INTRODUCTION

Arrays and matrices are one- or two-dimensional tables of contiguous numeric or string values. They are generally thought of in terms of rows and columns. Each array or matrix element is represented by a variable followed by a parenthesized integer value or values, called "subscripts". In a two-dimensional array element representation, the first subscript represents rows, the second, columns. Thus element A(2,2) is in row 2, column 2 of array A. If a non-integer subscript value is entered, BASIC/VM truncates it to an integer before using it to locate the specified element.

A matrix consists of those elements in an array that are represented by non-zero subscript values. In the figure below, array A has nine elements, while matrix A has only four. Note that all matrix elements are represented by non-zero subscripts.

The zeroth element in any array, that is, the foremost element, is represented as (0) in a one-dimensional array, and (0,0) in a two-dimensional array. A matrix, however, never includes a zeroth element; the first element in a matrix is (1) or (1,1), depending on the dimensions of the matrix. All the elements in an array are not printed if MAT PRINT is used to display the array; only the matrix portion of the array will be displayed. Similarly, only the matrix portion of an array will be operated on by the BASIC/VM MAT statements listed in Table 9-1.

The DIM statement dimensions 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:

(0,0)	(0,1)	(0,2)		
(1,0)	(1,1)	(1,2)	(1,1)	(1,2)
(2,0)	(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, that is, two rows by two columns. The actual dimensions of array A are 3 by 3, or 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) name elements in one-dimensional arrays. The values of all elements in a numeric array are initialized to 0 at the beginning of the program in which they are defined. Numeric array elements can be assigned values in a number of ways, including simple assignment statements like B9(6)=12, and the INPUT, READ and DATA statements discussed in Section 5.

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 subscript values. For example, the following subscripted variables represent string array elements:

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

All string array elements are variable length character strings; each element is initialized to null at the beginning of program execution. Simple assignment statements, like A$(2)= 'Steve', or the INPUT, READ and DATA statements, are used to assign new values to string array elements.

Declaring an array

Array dimensions are established in one of two ways: by a DIM statement, such as DIM A(5); or by a MAT statement, such as MAT PRINT A, which references array A. In the absence of a DIM statement, any referenced matrix (array) is implicitly assigned the default dimensions of (10) or (10,10). The elements in matrix A are A(1)-A(10). The value of any subscript must be within the range of the defined array dimensions. DIM statements may appear anywhere in a program. Before execution begins, BASIC/VM sets up arrays internally according to these criteria:

- If an array element, such as A(1) or A(2,3), is referenced in a program, but the array A has not been defined in a DIM statement, it is implicitly dimensioned to (10) or (10,10).
- If an array is defined more than once by DIM, the first dimension statement sets its size.

Default array dimensions: The following program references an array which has not been previously dimensioned by a DIM statement. Lines 20 and 30 assign values to two elements in this array. All elements have a value of 0, as indicated below:

```
10 ! Program with undefined array
20 A(3)=3  ! ASSIGN VALUE OF 3 TO THIRD ELEMENT OF ARRAY 1
30 A(4)=4  !ASSIGN VALUE OF 4 TO ELEMENT A(4)
40 REM MATRIX AUTOMATICALY DIMENSIONED TO 10
50 MARGIN 20  ! PRINT OUT MATRIX A IN SINGLE COLUMN
60 MATPRINT A
70 END
>RUNNH
0
0
3
4
0
0
0
0
0
0
```

The default dimensions of array A are assumed to be 10 because the array was not defined by a DIM statement.

Array examples: Below are several examples of one- and two-dimensional array and matrices. These arrays are defined by DIM statements and are assigned values by simple assignment statements using FOR-loops.

One-dimensional array:

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

Two-dimensional array: (Note that line 80 prints *matrix* M, not *array* M.)

```
 10 DIM M (3,4)
 20 FOR I = Ø TO 3
 30 FOR J = Ø 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 = Ø TO 3
110 FOR G = Ø TO 4
120 PRINT M (F,G)
130 NEXT G
140 NEXT F
150 PRINT LIN (2); 'DONE'
>RUNNH
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
10
9
8
7
6
```

Two-dimensional string array:

```
10 DIM B$(2,2)
20 B$(1,1)='CYNTHIA'
30 B$(1,2)='MICHELLE'
40 B$(2,1)='SABRA'
50 B$(2,2)='SHEILA'
55 MATPRINT B$
60 END
>RUNNH
CYNTHIA              MICHELLE
SABRA                SHEILA
```

Converting strings to arrays

A string of ASCII characters can be converted to a one-dimensional numeric array with the CHANGE statement. The resulting array contains the decimal equivalents of the characters in the string, including parity. Conversely, a decimal array may be changed to a string of corresponding ASCII characters. The CHANGE statement performs both types of data conversions, as shown in the example below. (Refer to Appendix B for the ASCII character table.)

The following program changes the ASCII characters in string D$ to an array containing 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) is automatically set to 6, because "CHANGE" is six characters in length.

The following program converts the decimal values of an array to a string of corresponding ASCII characters:

```
 5 A(Ø)=6 !Initialize A(Ø)
1Ø FOR X = 1 TO 6
2Ø READ A(X)
3Ø NEXT X
4Ø DATA 2Ø8, 21Ø, 197, 211, 212, 2Ø7
5Ø CHANGE A TO A$
6Ø PRINT A$
```

When run, the program prints:

```
PRESTO
```

Changing string length: Changing the zeroth element of a numeric array during a CHANGE alters the length of the resulting character string. For example, if A$="ABCDE" is converted to its numeric array equivalent, the first array element, A(0), will be equal to 5 because the string is five characters long. If the value of A(0) is set to 3, a CHANGE of A to A$ will result in a string of three characters: "ABC".

MATRICES

Dimensioning a matrix

Matrices, like arrays, are dimensioned by the DIM statement or are automatically defined when referenced by a MAT (matrix) statement. An unofficially dimensioned matrix is assigned default dimensions of (10) or (10,10) when referenced in a program. The dimensions of a matrix can be increased or decreased by using the MAT statement followed by new subscript values for the parameters, (dim-1, dim-2). See Table 9-1 for matrix statement formats.

Assigning matrix element values

Matrix elements may be assigned values with the simple assignment statement, the MATREAD - DATA statement combination, and the MATINPUT statement.

MATREAD and DATA: Matrix elements can also be assigned values with MATREAD and DATA statements. Values are read from DATA statements with MATREAD and are assigned to each element of the specified matrix in row-major order. (The rightmost subscript is incremented most rapidly.) 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 is assigned a value, or until the data list is exhausted. The following example reads fifteen numbers from consecutive DATA statements and assigns them to matrix A:

```
1Ø MARGIN 5Ø
2Ø DIM A(3,3)
3Ø MATREAD A
4Ø DATA 1Ø,2Ø,3Ø,4Ø,5Ø,6Ø,7Ø
45 DATA 8Ø,9Ø,1ØØ,11Ø,12Ø,13Ø,14Ø,15Ø
5Ø MATPRINT A
55 MARGINOFF
6Ø END
>RUNNH
1Ø                20                3Ø
4Ø                5Ø                6Ø
7Ø                8Ø                9Ø
```

Entering data with MATINPUT: Matrix values may be entered directly from the terminal using the MATINPUT statement. MATINPUT and INPUT are identical except that the values entered with MATINPUT are stored internally in matrix format. Values may be entered with leading and/or trailing blanks, just as with INPUT. All expected values can be entered on one line providing they are separated by delimiters.

The following program defines matrix B as a 2 by 3 matrix, with a total of six elements. The MATINPUT statement expects six values from the terminal. Each entered value is then assigned to an element in matrix B. MATPRINT displays all the matrix elements, putting each in a different print zone. The MARGIN statement can be used to force MATPRINT to display matrix data in column format. The third example below illustrates the use of MARGIN in formatting matrix output.

```
5 PRINT 'ASSIGN VALUES FOR MATRIX B:'
10 DIM B(2,3)
20 MATINPUT B
30 MATPRINT B
40 END
>RUNNH
ASSIGN VALUES FOR MATRIX B:
!10
!20
!30
!40
!50
!60
 10             20             30
 40             50             60

>! All six values can be entered at once:
>RUNNH
ASSIGN VALUES FOR MATRIX B:
!10,20,30,40,50,60
 10             20             30
 40             50             60
```

Multiple matrix variables are allowed in MATINPUT and MATPRINT statements. For example:

```
10 dim a(2,3)
20 dim b(2,3)
30 matinput a,b
40 matprint a,b
50 end
>runnh
!1,2,3,4,5,6
!7,8,9,10,11,12
 1              2              3
 4              5              6

 7              8              9
 10             11             12
```

19.0

The MATINPUT mat (*) statement is a special version of MATINPUT. It accepts one line of input and automatically dimensions the matrix, named by **mat**, to the number of items input.

```
5 REM AUTOMATICALLY DIMENSIONS MATRIX TO NUMBER OF ITEMS INPUT
10 DIM A (2,2)
15 MATPRINT A
20 MATINPUT 'ENTER SOME VALUES FOR MATRIX A:', A(*)
30 MATPRINT A
40 MARGIN 20
45 MATPRINT A
50 END
>RUNNH
0                    0
0                    0

ENTER SOME VALUES FOR MATRIX A:10,20,30,40
10                   20                   30                   40

10
```

```
20
30
40
```

On the basis of the values entered for MATINPUT*, matrix A is automatically defined as a one-dimensional matrix containing four elements.

MATRIX OPERATIONS

Matrix operations are valid only for that part of an array defined as a matrix, that is, those elements which have 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.

Summary of matrix operations

The following table lists all available BASIC/VM matrix operations. The parameter **dim** 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 during scalar multiplication.

Table 9-1. Matrix Operations

Type	Statements Used
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. (Scalar multiplication.)	MAT A=(num-expr)*X
The dimensions of one matrix are assigned to another.	MAT A=B
Two matrices are multiplied.	MAT A=X*Y
All elements are transposed.	MAT X=TRN(Y)
A square matrix is inverted.	MAT X=INV(Y)
Data exchange within the program.	MAT READ, CHANGE
Data exchange between program and terminal.	MAT INPUT, MAT PRINT
Data exchange between program and external files or devices.	MAT READ #, MAT WRITE #

Initializing a matrix

A matrix can be assigned values with any one of four matrix functions. These matrix functions are ZER, CON, IDN and NULL. The NULL function applies to string matrices only; the remaining functions are used with numeric matrices.

ZER function: ZER initializes each element of the specified matrix to 0. The following statements define a 5 by 7 matrix and initialize each element to 0:

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

CON function: 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
```

IDN function: IDN initializes the matrix to the identity matrix, in which all elements except those on the main diagonal are 0, and the diagonal elements are 1. IDN can only be used on a square matrix, that is, one in which the number of rows equals the number of columns. For example:

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

results in:

    1 Ø Ø
    Ø 1 Ø
    Ø Ø 1
```

NULL function: NULL has the same effect on string arrays that ZER has on numeric arrays; it initializes each element of the matrix to a null value. The matrix can also be re-dimensioned. For example:

```
100 DIM A$(2,2)
110 FOR I=1 TO 2
120 FOR J=1 TO 2
130 A$(I,J)='LARRY'
140 NEXT J
150 NEXT I
160 MATPRINT A$
170 MAT A$ =NULL(3,2)
175 PRINT 'PRINT REDIMENSIONED MATRIX'
180 MATPRINT A$
190 END
>RUNNH
LARRY                LARRY
LARRY                LARRY

PRINT REDIMENSIONED MATRIX
```

Of course, nothing is printed because all the matrix elements have been NULLed.

Redimensioning a matrix

The matrix functions ZER, CON, NULL and IDN can also be used to change the dimension of the matrix. By specifying subscript values 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:

```
10 REM First dimension matrix A
20 DIM A(3,3)
30 MATPRINT A
40 REM Redimension matrix A
50 MAT A=CON(2,2)
60 MATPRINT A
70 END
>RUNNH
 0             0             0
 0             0             0
 0             0             0

 1             1
 1             1
```

Setting dimensions with expressions: Dimensions do not have to be set with constants. For example, numeric expressions can be used in setting dimensions for matrix A below:

```
100 DIM A(2,2)
110 MATPRINT A
120 I=3
130 J=2
140 MAT A=CON(I,J+1)
150 PRINT 'Redimensioned matrix looks like this:'
160 MATPRINT A
>RUNNH
 0             0
 0             0

Redimensioned matrix looks like this:

 1             1             1
 1             1             1
 1             1             1

STOP AT LINE 160
```

Assigning one matrix to another: The dimensions of a matrix can also be changed by assigning another matrix to it. For example, the dimensions of matrix B are assigned to matrix A:

```
DIM A(6,6)     !Total of 36 elements in matrix
DIM B(5,4)     !Total of 20 elements in matrix
MAT A = B      !A is a 5 by 4 matrix of 20 elements
```

Matrix addition The elements of two numeric matrices may be added together, and the resulting values assigned to elements in 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 dimensions of target matrix (A) are converted to those of matrices 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		30
34		12		22

Two-dimensional Matrices:

MAT C	=	A	+	B
26 22 6		14 20 3		12 2 3
20 18 26		19 4 7		1 14 19
45 28 13		8 23 10		37 5 3

Figure 9-1. Matrix Addition

Matrix subtraction Elements of one matrix may be subtracted from elements of another matrix, providing that both matrices are of the same dimensions. The resulting values can then be assigned to the elements of a third matrix, also of the same dimensions. In the expression:

```
MAT A = B - C
```

the elements of matrix A are set equal to the values resulting from the subtraction of elements in matrix C from those of matrix B. All three matrices have the same dimensions.

Matrix multiplication A matrix can be multiplied by a numeric expression, or by another matrix. Both types of multiplications and their restrictions are discussed below.

Scalar multiplication: A matrix may be multiplied by a numeric scalar expression and the results stored in a second, or target, matrix. This is known as "scalar" multiplication.

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.

Multiplication of two matrices: To multiply two matrices, both must be two-dimensional and the number of columns in the first matrix must equal the number of rows in 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.

```
10 dim B(2,3)
20 matinput B
30 print
40 print 'THIS IS MATRIX B'
50 matprint B
60 dim C(3,4)
70 matinput C
80 print
90 print 'THIS IS MATRIX C'
100 matprint C
110 mat A=B*C
120 rem Matrix A is a 2-by-4 matrix
130 print
140 print 'THIS IS MATRIX A'
150 matprint A
160 end
>runnh
!1,2,3,4,5,6

THIS IS MATRIX B
1                    2                    3
4                    5                    6

!1,2,3,4,5,6,7,8,9,10,11,12

THIS IS MATRIX C
1                    2                    3                    4
5                    6                    7                    8
9                    10                   11                   12


THIS IS MATRIX A
38                   44                   50                   56
83                   98                   113                  128
```

Each element in matrix A is the result of multiplying the elements of matrix B by the elements of matrix C in the following manner:

1. Multiply each element in row 1 of matrix B, by each element in column 1 of matrix 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 element A(2,2); row 2 and column 3 to produce element A(2,3).

A = B*C
A(1,1) = (1*1) + (2*5) + (3*9)
A(1,1) = 1 + 10 + 27
The first element in matrix A = 38

Restriction: The current values of the target matrix may not be used as part of the multiplication expression:

```
MAT A = A*B
```

is an ILLEGAL expression in BASIC/VM.

Inverting a matrix

Matrix inversion is accomplished by using the INV function. A matrix can be inverted only if it is square and its determinant, (DET), 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 10). More information on DET and other matrix operations can be found in the following references, or in any other Linear Algebra text:

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 program segment demonstrates the use of the inverse and determinant functions:

```
60 MAT READ A
70 IF DET A = 0  THEN PRINT "CAN'T INVERT" ELSE PRINT "CAN INVERT"
80 IF DET A <> 0  THEN MAT C= B*INV(A)
90 DATA 1,2,3,1
```

Transposing a matrix

Every two-dimensional matrix has a transpose. This is determined by rolling the matrix on the main diagonal. If matrix A looks like this:

4 2
3 1

its transpose looks like this:

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).

Restriction: It is not legal to assign the transpose of a matrix to itself. For example, the statement MAT A=TRN(A) is *illegal*.

Using transposition: If a company sells four products and has three customers buying varying amounts of each product, the quantities can be set up in a 3 by 4 matrix (A). See Figure 9-2. The cost of each product can 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 a matrix of dimension (3,4) by a matrix of dimension (1,4), you must transpose B. This will make the number of columns in matrix A equal to the number of rows in matrix B.

This can be expressed in one of several ways, including:

```
C=A*TRN(B)

    or

D=TRN(B)
C=A*D
```

In either case, the final multiplication performed is:

```
C=(3,4)*(4,1)
```

and the result is:

```
C=(3,1)
```

Matrix I/O to data files

Matrices can be written to and read from data files with the MAT READ and MAT WRITE statements. Only the file handling statements which deal with matrix I/O are presented here. Other file handling operations are dealt with in Section 8 and Appendix E.

Matrix A

Customer 1	100	100	60	0
Customer 2	0	0	300	10
Customer 3	50	100	100	100
	nuts	bolts	nails	screws

Matrix B

Cost/item	.10	.05	.01	.02
	nuts	bolts	nails	screws

```
C = A * TRN(B)
  = (3×4) * (4×1)
  = (3×1)
```

Matrix C

Customer 1	Customer 2	Customer 3
Owes	Owes	Owes
15.60	3.20	13.00

Figure 9-2. Matrix Transposition

Writing a matrix to a file: Matrices are written to a data file with the MAT WRITE #unit 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 opened data file with a single MAT WRITE statement. For example:

```
10 DEFINE FILE #1 = 'NUMBERS', ASCSEP
20 DIM A(3)
30 A(1)=10
40 A(2)=20
50 A(3)=30
60 MAT WRITE #1, A
70 REWIND #1
80 MAT READ #1, A
90 MATPRINT A
100 CLOSE #1
>RUNNH
10                    20                    30
```

Reading from a file to a matrix: Values written to a file with MAT WRITE can be retrieved with any of the READ statements covered in Section 8. The contents of a file record can also be read into one or more matrices with the MAT READ or MAT READ* statements, as shown in the program above. The matrix or matrices to be filled must first be defined by a DIM statement. Data is then read from the indicated file until all elements of the matrix or matrices have been assigned values.

The following example illustrates the use of MAT READ and MAT WRITE with three types of ASCII files. Three values are written to each file type using WRITE# statements. Matrix A is defined as a one-dimensional matrix of two elements. A MAT READ statement retrieves values from each of the files to fill the matrix. In this example, MAT READ reads only two values from each file, because matrix A has only two elements.

```
5 ON ERROR GOTO 200
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
200 END
>RUNNH
READ MAT
20                    20
```

```
READ MAT2

20                    20

READ ASCDA
20                    20
```

Reading from default ASCII files: Notice that the default ASCII file, "MAT", is handled somewhat differently from other ASCII file types. Literal delimiters must be included in the data written to the file: otherwise, the contents of the record will be inpterpreted as a single string item by the MAT READ statement. See line 40 in the program above. Because of these properties, default ASCII files are not as well-suited to matrix I/O as are other file types. See Section 8 and Appendix E for more details.

MAT READ* statement: The MAT READ* statement works just like the READ* statement introduced in Section 8. 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. For example:

```
10 DEFINE FILE #1 = 'T.MATREAD', ASCSEP
20 WRITE #1, 10,10,10,10,10
30 WRITE #1, 20, 20,20,20,20
40 REWIND #1
50 DIM A(3)
60 MATREAD* #1,A
70 PRINT 'FIRST MATREAD*'
80 MATPRINT A
90 MATREAD* #1, A
100 PRINT 'SECOND MATREAD*'
110 MATPRINT A
120 REWIND #1
130 MATREAD #1, A
140 PRINT 'FIRST MATREAD'
150 MATPRINT A
160 MATREAD #1, A
170 PRINT 'SECOND MATREAD'
180 MATPRINT A
190 END
>RUNNH
FIRST MATREAD*
10                    10                    10

SECOND MATREAD*
10                    10                    20

FIRST MATREAD
10                    10                    10

SECOND MATREAD
20                    20                    20
```

This program clearly shows the difference between MAT READ and MAT READ*. The first two matrices printed contain file values read with MAT READ* statements. The third and fourth matrices contain values read from the file with MAT READ statements.

10 Functions

INTRODUCTION

Most arithmetic operations can be simplified by using pre-defined numeric functions to handle routine calculations such as computing square roots and logarithms. String data handling can also be facilitated by functions designed specifically for manipulating strings. BASIC/VM provides both numeric and stringsystem functions to perform operations like calculating sine and cosine, generating random numbers, and converting a string of digits 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, a function can be used anywhere in an expression where a numeric constant or variable can be used.

The following table lists the library of BASIC/VM numeric system functions. In all of the descriptions, X represents a numeric expression, and Y and Z represent integers.

Using numeric system functions

Most of the functions in Table 10-1 perform familiar mathematical operations. Below are some of the ones most frequently used in BASIC programming.

The INT function: The INT function performs integer truncation and can be used to convert a decimal number to an integer. For example:

```
INT(.99989) = 0
```

INT can also be used to round any numeric value to a specific number of decimal places. For example, to round the value of X1 to one decimal place, use the formula:

```
INT  (10 * X1 + .5) /10
```

For example:

```
INT(2.9 + .5) = 3
```

Table 10-1 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 0 to π. Argument must be in the range: $-1 \leqslant X \leqslant 1$.
ASN(X)	Computes the principal arcsine of X. The result is in radians in the range of $-\pi/2$ to $\pi/2$. Argument must be in range: $-1 \leqslant X \leqslant 1$.
ATN(X)	Computes the principal arctangent of X radians. Returns the angle whose tangent equals X. Argument, in radians, is in the range of $-\pi/2$ to $\pi/2$.
COS(X)	Computes the cosine of X. The argument must be 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). The argument must be in radians.
DEG(X)	Computes the number of degrees in X radians, [(180/π)*X)]. The result is in degrees.
DET(X)	Computes the determinant of matrix X. If DET(X) unequal to 0, 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 the exponential of X, defined as "e raised to the X power". The value of e is: 2.71828...
INT(X)	Performs integer truncation. If $X \geqslant 0$, returns the greatest integer $\leqslant X$. If $X < 0$, returns the least integer $> -X$.
LIN#(X)	In ASCLN files, returns the line number of the current record on unit X. For BINDA files, returns the current record positioned to in the file on unit X. Top of file is record 0.
LOG(X)	Computes the natural logarithm (base e) of X.
NUM	Returns the actual number of entries supplied to the MATINPUT M(*) and MATINPUT M$(*) statements. Matrix M is one-dimensional.
PI	Computes the value of π (3.14159).
RAD(X)	Computes the number of radians in X degrees. Formula is: X*(π/180).
RND[(X)]	If $X > 0$, uses X to initialize the "random" number generator and returns X as the function value. If $X < 0$, uses X to initialize the random number generator, and returns a value in the range zero to one. If X=0, returns a random number between 0 and 1.RANDOMIZE used to reset generator to new starting point in random number series.

Table 10-1. (cont'd)

SGN(X)	Computes a value based on the sign of X as follows: X < 0 SGN(X)=–1 X=0 SGN(X)=0 X > 0 SGN(X)=1
SIN(X)	Computes the sine of X. The argument must be 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. Argument must be in radians.
SQR(X)	Computes the positive square root of X.
TAN(X)	Computes the tangent of X. The argument must be in radians.
TANH(X)	Computes the hyperbolic tangent of X defined as [(EXP(X)–EXP(–X)]/[EXP(X)+EXP(–X)]]. The argument must be in radians.

To round X1 to two decimal places, use the following formula:

```
INT  (100 * X1 + .5) /100
```

For example:

```
INT(100 * 39.456 + .5)/100 = 39.46
```

The random number generator: The RND function generates a series of pseudo-random numbers. Given identical starting conditions, the number "randomly" chosen as the starting point will be the same each time the program is run. The range of "random" numbers returned depends on the value supplied for X. RND(0) initializes the number generator and returns a result in the range of 0 to 1 (exclusive). RND(X), where X > 0, returns the function value of X; RND(–X) returns a result in the range of 0 to 1.

The following program yields ten random integers of three digits or less. The INT function is used to convert the very small decimal values yielded by the RND function to integers.

```
10 FOR I=1 TO 10
30  L=RND(0)
35  L1=INT(L*1000)
40  PRINT L1
50 NEXT I
60 END
>RUNNH
352
301
216
173
676
85
7
858
668
873
```

The RANDOMIZE statement: To guarantee a more "random" set of numbers, include a RANDOMIZE statement prior to the occurrence of the RND function in the program. RANDOMIZE initializes the random number generator to a new and unpredictable starting location in the number series each time it is used. RND(0) and RANDOMIZE are equivalent in function.

The following example shows how RANDOMIZE works in program that uses the random number generator. The first set of program executions illustrates what often happens when the number generator is reset to the same point in the series during program execution.

```
10 PRINT 'RND(0) IS':RND(0)
20 X=1
30 PRINT 'RND(1) IS':RND(X)
40 X=0
50 PRINT 'RND(0) IS':RND(X)
60 END
>! First run:
>RUNNH
RND(0) IS .07571411132813
RND(1) IS 1
RND(0) IS .7112731933594
>!Second run:
>RUNNH
RND(0) IS .9403991699219
RND(1) IS 1
RND(0) IS .7112731933594
>!Third run:
>RUNNH
RND(0) IS .3080444335938
RND(1) IS 1
RND(0) IS .7112731933594
```

When the RANDOMIZE statement is inserted after line 40, the following results are observed:

```
RND(0) IS .7257080078125
RND(1) IS 1
RND(0) IS .9139404296875
>RUNNH
RND(0) IS .1993103027344
RND(1) IS 1
RND(0) IS .3198852539063
>RUNNH
RND(0) IS .6022338867188
```

Observe that all the values returned for RND(0) are different after the RANDOMIZE statement is added to the program.

More system functions

Most of the numeric functions in the BASIC/VM library are easy to use and require little or no explanation. Restrictions on the argument values supplied to these functions are noted in Table 10-1.

Some common uses for these numeric functions are: calculating the number of radians in a given angle (RAD), and calculating the sine (SIN), cosine (COS) and tangent (TAN) of an angle.

```
10 REM THIS PROGAM CALCULATES THE SIN,COS,TAN
20 REM OF AN ANGLE
30 INPUT 'HOW MANY DEGREES IN ANGLE A?', X
35 PRINT
40 PRINT 'ANGLE A IS ':X:'DEGREES'
50 PRINT
60 Y=RAD(X)
70 PRINT 'RADIANS IN ANGLE A:':Y
80 X1=SIN(Y)
90 X2=COS(Y)
100 X3=TAN(Y)
110 PRINT
120 PRINT 'SIN OF A', 'COS OF A', 'TAN OF A'
130 PRINT
140 PRINT X1,X2,X3
150 PRINT
160 END
>RUNNH
HOW MANY DEGREES IN ANGLE A?45

ANGLE A IS  45 DEGREES

RADIANS IN ANGLE A: .7853981633975

SIN OF A              COS OF A              TAN OF A

.7071067811865        .7071067811865        1
```

The next program uses several other numeric functions like ABS, EXP, SQR and LOG to find the absolute value, exponential representation, square root, and logarithm of several numbers:

```
10 REM THIS PROGRAM USES: ABS, EXP, LOG, SQR
15 !
20 INPUT 'ENTER THREE NUMERIC VALUES:', A,B,C
30 Y = SQR(A+B+C)
35 PRINT 'THE VALUE OF EXPRESSION Y :':Y
40 PRINT
45 PRINT 'THE ABSOLUTE VALUE OF Y IS:':ABS(Y)
50 PRINT
55 PRINT 'THE SIGN OF Y IS:':SGN(Y)
60 PRINT
65 PRINT 'LOG', 'EXP', 'SQR'
70 PRINT
72 REM  USE INTEGER VALUES OF A AND B
75 A=ABS(A)
80 B=ABS(B)
85 PRINT LOG(A), EXP(A), SQR(A)
90 PRINT
95 PRINT LOG(B), EXP(B), SQR(B)
100 END
>RUNNH
ENTER THREE NUMERIC VALUES:123,45.6,100.89
THE VALUE OF EXPRESSION Y : 16.4161505841
```

```
THE ABSOLUTE VALUE OF Y IS: 16.4161505841

THE SIGN OF Y IS: 1

LOG                    EXP                    SQR

4.812184355372         2.619517318744E+53     11.09053650641

3.81990771652          6.365439207171E+19     6.752777206454
```

ACS and ASN restrictions: The arccosine and arcsecant functions require the supplied argument to be in the range of –1 to +1. If the argument is out of range, an error message occurs:

```
5 ! ARG FOR ACS AND ASN MUST BE -1<=X<=1
10 X=2
20 X2=1
30 X3=-1
40 X4=0
50 PRINT 'ACS' , 'ASN'
60 PRINT
70 PRINT ACS(X2), ASN(X2)
80 PRINT ACS(X3), ASN(X3)
90 PRINT ACS(X4), ASN(X4)
100 PRINT ACS(X), ASN(X)
110 END
>RUNNH
ACS                    ASN

0                      1.570796326795
3.14159265359          -1.570796326795
1.570796326795         0
ASN,ACS - ARGUMENT RANGE ERROR AT LINE  100
```

STRING SYSTEM FUNCTIONS

BASIC/VM provides a substantial library of string functions, enabling the programmer to manipulate strings in a variety of ways. 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$,Y,Z), returns a substring, beginning with character Y through character Z, of a larger string, X$. String functions, like STR$(X), can also convert a numeric item to its corresponding string representation. They can also convert the string representation of a number to the numeric value it represents, for instance, VAL(X$), where X$='12,456.34'.

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, Y and Z represent any numeric expressions, and X$ and Y$ represent a string expression.

Using string functions

Some of the string functions listed in Table 10-2 are discussed in the following paragraphs. Several examples are included to illustrate the properties of these functions.

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 Y is not used and the operation is unsuccessful, a run-time error occurs.

Case conversions: The CVT$$ function permits both upper-to-lower and lower-to upper case conversions for string constants. The appropriate masks for conversion are listed in Table 10-3. Both types of case conversion can be performed simultaneously on a string containing upper- and lower-case letters. For example:

```
If X$='Big Deal':

   CVT$$(X$,32):  returns 'BIG DEAL'

   CVT$$(X$,256):  returns 'big deal'

   CVT$$(X$,256+32):  returns 'bIG dEAL'
```

The last example shows how masks can be combined additively to perform both conversion operations at once.

VAL and STR: The functions STR and VAL ignore leading and trailing blanks in the arguments supplied them. The program below illustrates this property.

```
5 REM No leading or trailing blanks
6 REM output in results of VAL or STR functions
```

```
10 X=.456789
20 PRINT 'STR$(X) IS':STR$(X)
30 X$='  123.45  '
40 PRINT 'VAL(X$) IS:':VAL(X$)
50 PRINT
60 Y$='$123.45'
70 PRINT VAL(Y$)
80 ! NON-NUMERIC CHARACTERS MAY NOT BE INCLUDED IN STRING Y$
90 END
>RUNNH
STR$(X) IS .456789
VAL(X$) IS: 123.45

VAL ARG NOT NUMERIC  AT LINE  70
```

Table 10-3. Masks For CVT$$

Mask	Function
1	Forces parity bit off.
2	Discards all spaces.
4	Discards .NUL. ,.NL. ,.FF. ,.CR. ,.ESC..
8	Discards leading spaces.
16	Reduces multiple spaces to one space.
32	Converts lower case to upper case.
64	Converts [to (and] to).
128	Discards trailing spaces.
256	Converts upper case to lower case.

(Masks can be combined additively.)

Other string functions: The following example uses the LEN, SUB, RIGHT and other string system functions to manipulate a given string:

```
70 PRINT
80 PRINT 'Position of K in string:':INDEX(X$,'K',1)
90 PRINT
100 B$ = SUB(X$,21,28)
110 PRINT 'Substring in positions 21-28 is:':B$
120 PRINT
130 PRINT 'Rightmost characters B$:':RIGHT(B$, 3)
140 PRINT
150 PRINT 'ASCII code of letter H:': CODE('H')
160 PRINT
170 REM  Convert HUSBAND to lower-case letters:
180 PRINT 'Converted form of string B$:':CVT$$(B$,256)
190 END
```

```
10 REM Using string functions
20 X$='SOMEBODY KILLED HER HUSBAND'
```

```
160 PRINT
170 REM  Convert HUSBAND to lower-case letters:
180 PRINT 'Converted form of string B$:':CVT$$(B$,256)
190 END
```

When run, the program results in the following output:

```
String X$ is: SOMEBODY KILLED HER HUSBAND

Length of string X$: 27

Position of K in string: 10

Substring in positions 21-28 is: HUSBAND

Rightmost characters B$: SBAND

ASCII code of letter H: 200

Converted form of string B$: husband
```

USER-DEFINED FUNCTIONS

In some programs it is necessary to execute the same sequence of statements or mathematical operations in several different places. BASIC/VM allows you to define your own functions and use them just like system functions. These functions can be kept in separate files and CHAINed to or LOADed into other programs as needed.

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 where the function is called or referenced in the program. For 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 numeric function may be a numeric or string expression. The expression is evaluated, and then the value of the expression is substituted for the argument in the function definition. For example:

```
LET A1 = 3.14 * FNA (X1 + COS(B))
```

A user function may also be longer than one line. After the last line of the function definition, use the FNEND statement to indicate the end of the function definition.

```
 .
 .
 .
100   DEF FNA (I)
110   IF I=0 THEN FNA=-1 ELSE FNA = I*I+I
120   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

The name of a string user-defined function consists of the letters FN followed by a simple string variable, as shown below:

FNA$** or **FNA7$

A user-defined function must be defined by a DEF statement. If the definition is longer than one line, 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 is transferred to the lines which defined the function. For 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   INPUT A


 .
 .
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 control hinges on the value for A entered at line 70. Assuming A=1, control jumps from line 100 to line 1000. At line 2000, control transfers to line 5000. When the function FNA$ is referenced in line 2500, control transfers to lines 30, which begins the function definition of A$. When line 60 is reached, control then returns to continue evaluation of the expression which contained the function reference (line 5000).

User-defined string functions may also be used in conjunction with system functions:

```
DEF FNF$ (A,B,C) = LEFT (STR$ (A+B+C),5)
```

Here, both the string functions LEFT and STR$ are employed in the definition of the function FNF$.

Defining functions

A function in BASIC/VM is defined with a parameter; for example, in the statement: DEF FNA(X), X is the parameter. The parameter identifies a variable or an array, and is sometimes referred to as a "dummy" variable.

A function is called in a program by an expression consisting of the function name, (for example, FNA) followed by a parenthesized argument or set of arguments.

DEF FNA(X)	**X** is a parameter of function FNA.
x=5	**X** is a dummy variable set equal to 5.
FNEND	Denotes end of function.
.	
.	
.	
Y=10	**Y** is an external program variable.
Z=FNA(Y)	This expression calls function **FNA**; **Y** is argument which shares a 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=10

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 (10) 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 10. 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.

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. There is a direct correlation between the argument 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 X^2.
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
```

USING USER-DEFINED FUNCTIONS

There are many ways in which user-defined functions can make BASIC/VM programming easier and more efficient. The remainder of this section provides additional information regarding the implementation of user-defined functions in BASIC/VM. In particular, the following topics are discussed:

- Function definition and program control
- Avoiding function-I/O interaction
- "Local" variables and arrays: the LOCAL statement
- Enhancing modular programming
- Simple cursor-positioning (for CRT's)

Function definition and program control

The following program demonstrates how program control can be directed by using functions and subroutines. The function FNA is used in association with a "subroutine" that begins in line 5000. Remember that a function definition must always precede the program statement that calls or references it.

```
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 INPUT A
 .
 .
 .
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. When the function is referenced (in line 5000), control transfers to lines 30 through 60 where the function is defined. Control then returns to line 5000 for the assignment and continues sequentially. After line 6000, control returns to 2010.

Recursive capability: Functions may also be recursive that is, 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

There are some noteworthy cautions regarding the placement of function calls, function definitions and control transfers within the same program. For instance, do not transfer program control into or out of a function definition. This can cause system stack difficulties leading to unpredictable behavior. Additionally, when dealing with functions that perform I/O operations, like READs, WRITEs, PRINTs and INPUTs, avoid calling these functions within other I/O statements. It may be confusing to the I/O handler to call a function to do a READ, for instance, while the function is being printed.

In the following example, the function FNI$ performs a file READ in both programs. In the first program, WRONGPROG, FNI$ is placed on the I/O list and is called to do a file READ while its value is being printed: PRINT FNI$(A). The second program, RIGHTPROG, solves the problem of potentially ambiguous I/O by assigning the information returned by the function READ to a temporary variable, T$. This information will not be intermixed with the actual printout of the function.

```
WRONGPROG
10 ! THIS IS WRONGPROG
20 ! THIS PROGRAM MAY CONFUSE THE I/O HANDLER
30 !
55 ! DEFINE READ FUNCTION: FNI$
60 DEF FNI$(A)  !READS A STRING FROM FILE UNIT A
70 !
75 READLINE #A,X$
80 FNI$ = X$
90 !
100 !
110 FNEND
120 ! READ FILE 'YYY', PRINT ON TERMINAL
130 A=1
140 DEFINE FILE #A = 'YYY'
150 FOR I = 1 UNTIL 1=2
160 PRINT FNI$(A)
170 ! NO TEMPORARY VARIABLE ASSIGNED TO FUNCTION READ
180 NEXT I
```

```
RIGHTPROG
10 ! THIS IS RIGHTPROG
20 ! THIS PROGRAM DOES NOT CONFUSE THE I/O HANDLER
30 !
55 ! DEFINE READ FUNCTION: FNI$
70 !
75 READLINE #A,X$
80 FNI$ = X$
90 !
100 !
110 FNEND
120 ! READ FILE 'YYY', PRINT ON TERMINAL
130 A=1
140 DEFINE FILE #A = 'YYY'
150 FOR I = 1 UNTIL 1=2
160 T$ = FNI$(A)  !ASSIGN VALUE RETURNED BY READ FUNCTION
170               !TO A TEMPORARY VARIABLE
180 PRINT T$
190 NEXT I
```

Local variables and arrays

The LOCAL statement enables both variables and arrays to be defined as *local* (as opposed to global) in a function definition. LOCAL should appear immediately after the DEF statement which indicates the beginning of a function definition. Variables or arrays to be considered local to this function should not be referenced in the definition until after the LOCAL statement is issued.

Defining local variables: Variables are declared local to a function by listing them in a LOCAL statement, as in the example below:

```
10 DEF FNP$(X$,Y$,N)! pads X$ on right with character Y$.
20   LOCAL Z$
```

```
30    Z$ = X$
40    Z$ = Z$ + Y$ UNTIL LEN(Z$) = N
50    FNP$ = Z$
60 FNEND
```

The variable Z$ takes on the values indicated in the function definition. These values are retained over multiple calls to the function; that is, they are static variables. Z$ may appear elsewhere in the program as a global variable and will be treated as a totally different entity. Global and local variables of the same name can co-exist without affecting one another.

Note that local variables, like function arguments, cannot be PRINTed in immediate mode during a PAUSE or BREAK.

Defining local arrays: Arrays are defined as local in the same manner as are variables. The example below shows the definition of local variables (line 20) and local arrays (line 30).

```
10 DEF FNA(X)
20    LOCAL I, J
30    LOCAL DIM A(10,10)
40    A(I,J) = SIN(X*I*J) FOR I = 1 TO 10 FOR J = 1 TO 10
50    FNA = DET(A)
60 FNEND
```

Enhancing modular programming

The LOCAL feature makes modular programming easier in BASIC/VM. Users can keep libraries of functions for use in several different programs. Simply RESEQUENCE these function definitions before LOADing them into the desired program. The need to search for variable-name conflicts is effectively eliminated by the LOCAL feature. Variables not defined as "local" are assumed to be "global". Thus the same variable, for instance, F, can be used in both a local and global capacity in the same program without confusion.

Cursor positioning

During a PRINT operation, data can be displayed at any position on the terminal screen. This is accomplished by moving the cursor to the desired screen location at which the item is to be PRINTed.

Simple cursor positioning can be achieved in BASIC/VM by using the special "@" character. "@" is a shorthand notation for the user-defined function, FNZ9$, which may be used to define cursor control on a terminal screen.

With this function you can PRINT an item at a particular terminal screen location, by typing PRINT, followed by: the @ character, and the parenthesized row and column coordinates, (row, column) of the desired screen location.

For example, to position the cursor to row 12, column 4 during a PRINT operation, type:

```
PRINT @ (12,4);  'string'
```

The cursor positions to row 12, column 4, and the indicated string, 'string', is PRINTed at this screen position. Note, however, that this is just one method of defining a cursor control function.

Cursor control functions: Cursor positioning requires that you be familiar with your terminal's cursor positioning capabilities, as well as the control characters and codes needed for cursor control. See the manual that came with your particular terminal for a list of control codes. Each terminal type requires a different function definition to control cursor positioning. However, most functions will be similar to the one listed below.

Sample control function: The function shown below sets up cursor positioning for a FOX terminal. Note that all margin checking should first be inhibited by typing: MARGIN OFF. This eliminates any difficulties BASIC/VM may have in achieving the indicated column position due to margin restrictions.

```
100 Margin off   ! release margin restrictions
110 ! Cursor-positioning function for FOX and OWL terminals
120 !
130 ! (r,c) are row, column coordinates
140 ! char(155) is Escape and char(159) is Unit Separator:
150 ! when followed by the x and y keys respectively,
160 ! these sequences control cursor position on the
170 ! FOX and OWL terminals
180 !
190 Def @(r,c)=char(155)+'x'+char(159+r)+char(155)+'y'+char(159+c)
200 !
210 ! This function tells BASIC how to do cursor positioning
220 ! given values for r and c
230 !
240 Print @ (12,30);  'ABCDE'
250 !
260 ! This prints ABCDE at row 12, column 30 on a
270 ! FOX or an OWL screen
280 !
```

11
Expressions

Sample control function: The function shown below sets up cursor positioning for a FOX terminal. Note that all margin checking should first be inhibited by typing: MARGIN OFF. This eliminates any difficulties BASIC/VM may have in achieving the indicated column position due to margin restrictions.

```
100 MARGIN OFF
110 REM Cursor-positioning function for FOX terminal
120 !
130 DEF @ (X,Y)=CHAR (155)+'X'+CHAR (32X)+CHAR+'Y'+CHAR (32+Y)
140 !
150 PRINT @ (12,30);  'ABCDE'
```

This function enables the printing of string 'ABCDE' in row 12, column 30 on a FOX screen.

11 Expressions

INTRODUCTION

Expressions can generally be defined as combinations of operands and operators in some form that can be evaluated. Expressions are a fundamental part of the BASIC/VM language structure. For example, it would be impossible to establish complex conditions for program control transfer without them. This section explains all the rules and regulations governing the formation and evaluation of expressions in BASIC/VM. There are four types of expressions:

- Logical
- Numeric (arithmetic)
- Relational
- String

Expressions are generally evaluated to yield a single result. Evaluation protocol is discussed first, as it applies to all expression operations.

EVALUATION PRIORITY LIST

All types of expressions in BASIC/VM are governed by the same evaluation rules. These rules dictate the order in which the individual components of an expression are to be evaluated. Evaluation is performed according to an established operator priority. This list indicates which operators have precedence over which, so that misinterpretation of a complex expression is unlikely.

The priority list is:

Expressions in parentheses
System and user-defined functions
^(or **)
NOT, unary (+,–)
*,/,MOD
+,–
MIN,MAX
relationals (= , < , > , < = , > = , < >)
AND
OR

Parenthetical expressions are always evaluated first, regardless of the operations they contain. Operations within parentheses are performed in order of priority. Then, the rest of the expression is evaluated, with operators of higher precedence being evaluated before operators of lower precedence, as dictated by the priority list. Operators at the same priority level are evaluated in left-to-right order, as they appear in the expression.

NUMERIC EXPRESSIONS

Numeric (arithmetic) expressions can consist of any combination of numeric constants, numeric variables, numeric array references and arithmetic operations. For example:

```
G2(Q1/Q2)
RND(1) + COS(Q)
A(2) + A(3) + A2
```

are all numeric expressions.

Arithmetic operators

The arithmetic operators used in forming numeric expressions are listed below.

Operator	Meaning	Example
	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

Evaluating numeric expressions

Numeric expressions are evaluated according to operator priority, as listed in Table 11-1, above. Parts of expressions enclosed in parentheses are evaluated first, from left to right, as they occur in the expression. Operators at the same priority level are evaluated in the order in which they appear; again, from left to right.

For example, consider the following simple numeric expression:

```
E = (A + B) /2
```

The addition operation, (A + B), being within parentheses, is performed first; then, the division by 2 is performed, even though the division operator has higher priority. The final result is then assigned to E. Remember that operators with equal priority are evaluated from left to right.

A more complex numeric expression, such as the one below, would be difficult to tackle without precedence guidelines. This expression:

```
A + B - C * D * E ^  F ^  G
```

is interpreted as:

```
(A + B) - ((C * D) * ((E^F)^G))
```

Evaluation of this expression occurs in the following steps, according to operator priority:

1. parenthetical expressions, in the order they appear:
   a. (A+B)
   b. (C*D)
   c. (E^F)

2. the exponential operations, in two steps:
   a. (E^F)
   b. (E^F)^G
3. result of C*D multiplied by result of ((E^F)^G)
4. the result of A+B minus the result of step 3

Note that evaluation of exponential expressions has been changed to conform with ANSI standards. The expression A^B^C is now evaluated as (A^B)^C; originally it was evaluated as A^(B^C).

STRING EXPRESSIONS

String expressions may be composed of various combinations of string constants, string array references and string function references, joined by the string operator. This operator, "+", is also known as the "concatenation" operator. Concatenation means appending one string to another as in:

```
10 A$= "CORN"
20 B$= "FLOWER"
30 C$=A$+B$
40 PRINT C$
50 STOP
>RUNNH
CORNFLOWER
STOP AT LINE 50
```

It is illegal to combine numeric and string operands in a string expression. Numeric values cannot be concatenated to string items. For example, the expression, "D$=A+B$", is ILLEGAL.

RELATIONAL EXPRESSIONS

Relational expressions are composed of numeric or string operands and relational operators. For example, A > B or A$ < =B$. String operands cannot be compared with numeric operands.

Relational operators

Relational operators are used to compare two or more numeric or string items. Below is a list of 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

Evaluating Relational Expressions

All relational operators are at the same priority level, just below the MIN and MAX operators in the priority list. They are evaluated in the order in which they appear in an expression.

Evaluating numeric relationals: Numeric relational expressions are evaluated on a strictly numeric basis. For example:

```
20 IF A>B GOTO 100
```

The "GOTO 100" clause will be executed only if the value of A exceeds the value of B. Otherwise, the next sequential statement will be executed.

In the statement:

```
10 IF A>=(B+C/D) GOTO 40
```

the expression (B+C/D) is evaluated and compared with the value of A. Control will be transferred to statement 40 only if the value of A is greater than or equal to the value of (B +C/D).

Relational expressions may also be written in the form:

```
IF A THEN GOTO 40
```

The sub-expression, "IF A", is interpreted by the compiler as: "IF A < > 0". An expression is considered "true" if it evaluates to a non-zero value. Conversely, an expression that evaluates to zero is considered "false". Therefore, if A is true, that is, if it does NOT evaluate to zero (0), then the transfer to statement 40 will take place.

Evaluation of string relational expressions: Comparison of values in string relational expressions is done on a character-by-character basis. Characters are ranked by alphabetical order. Each letter has a corresponding ASCII code, with "A" having a value of 193, and "Z", a value of 218. Characters in the beginning of the alphabet have lower decimal values and less rank than those towards the end of the alphabet. 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 first non-common character is reached in both strings. At this point, a decision is made on the basis of the relative alphabetical rank of the two characters being compared.

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).

Below are additional examples of string relational expression evaluation. The first example compares two strings, "MICHELOB" (A$) and "MILLER" (B$). The first two letters of the strings are identical, so comparison is done on the third character of each string.

```
05 PRINT "THIS IS A COMPUTER TASTE-TEST"
10 A$="MICHELOB"
20 B$="MILLER"
30 IF A$>B$ GOTO 55
40 PRINT "MILLER IS GREATER THAN MICHELOB"
50 GOTO 60
55 PRINT "MICHELOB IS GREATER THAN MILLER"
60 END
>RUNNH
THIS IS A COMPUTER TASTE-TEST
MILLER IS GREATER THAN MICHELOB
```

Lower-case letters have a higher decimal value than upper-case letters. For example:

```
10 A$="bad"
20 B$="BAD"
30 IF A$=B$ THEN PRINT "EQUAL"
35 IF A$>B$ THEN PRINT "A$ GREATER" ELSE PRINT "A$ LESSER"
40 END
>RUNNH
A$ GREATER
```

If strings are to be compared on the basis of physical length and not relational value, use the LEN function, as in:

```
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, as always. 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.

LOGICAL EXPRESSIONS

Logical expressions usually consist of one or more relational expressions and are joined by logical operators. Both numeric and relational expressions appear in the same logical expression. Logical expressions are evaluated according to the familiar "Truth Table", represented in Figure 11-1. Logical expressions are generally used with one of these keywords to determine program flow control: IF, WHILE, UNLESS or UNTIL.

Logical operators

Below is a list of logical operators, their meanings and an example of each in use:

Operator	Meaning	Example
AND	Logical "and"	I=J AND K$=L$
OR	Logical "or"	I=J OR I=K
NOT	Logical complement	NOT I

The three logical operators, AND, OR, NOT are used to combine relational expressions. They determine whether a statement is ultimately 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.

The result of a logical expression evaluation (true or false) is used to determine the flow of control within a program. Many of the examples in Section 6 use complex expressions to establish conditions for control transfer. For example, the following simple program shows the combination of relational expressions and logical operators to set up a logical expression, or condition, upon which control transfer will be based.

```
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 entered at line 10. If the expression is true, a GOTO is executed. If false, the next sequential statement is executed, and so on.

Evaluation of logical expressions

In a complex logical expression, each part is evaluated according to the operators present; then the entire expression is evaluated to a single result. The logical operators AND, OR and NOT determine whether the entire expression is false or true. The NOT operator ranks just below exponentiation on the priority list; AND and OR are the lowest operators on the priority list, and are therefore the operations to be performed in the evaluation process.

Given the values:

E=0
A=6
C=3
B=2
D=7

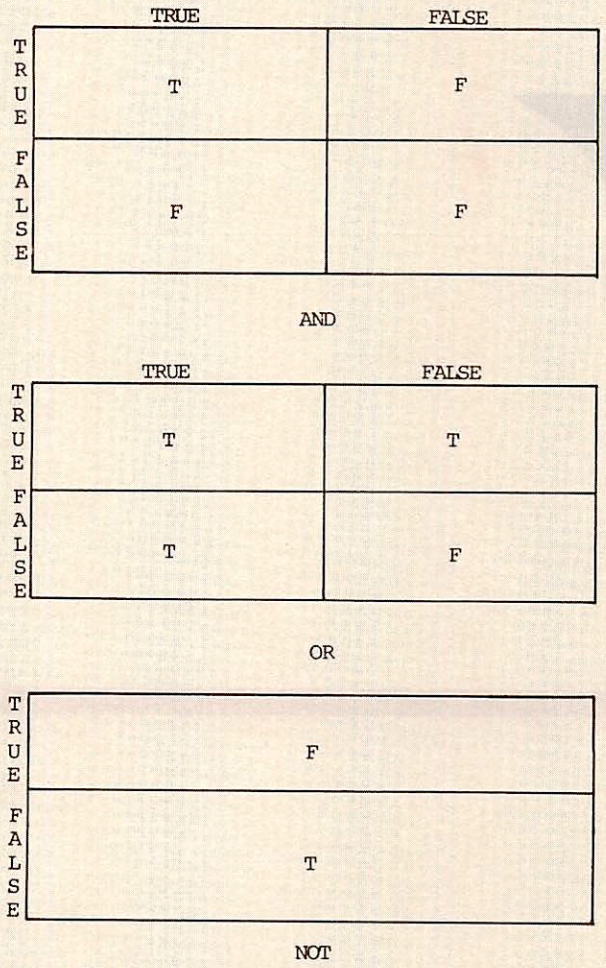


Figure II-1. Truth table for logical expressions

The expressions below would be evaluated as indicated:

E AND A-C/3	Evaluates to FALSE since the first term in the expression is false.
A+B AND A*B	Evaluates to TRUE since both terms in the expression are true.
A=B OR C=SIN(D)	Evaluates to FALSE since both expressions are false.
A OR E	Evaluates to TRUE since one term of the expression (A) is true.
NOT E	Evaluates to TRUE since E=0.

Note

Logical expression values can only be used in statements containing the keywords: If, WHILE, UNLESS, or UNTIL. Expressions such as "LET A=B < > C", or "A=B AND C", are ILLEGAL.

V

REFERENCE

12 PRIMOS commands

INTRODUCTION

This section summarizes all the PRIMOS commands referenced in this book. PRIMOS commands may be entered in upper- or lower-case letters. Abbreviations appear in rust-colored letters. All applicable conventions are listed in Section 2 of this guide.

PRIMOS COMMANDS

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.

CLOSE { x1 [,...xn] | ALL }

Closes file units specified by **x1** through **xn** . ALL option closes all file units (except COMO file units) currently opened by user or system. A list of open file units can be obtained with the STATUS UNITS command.

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 { -CONTINUE | -END | -PAUSE | pathname | -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 { CONTINUE / -END / -NTTY / -PAUSE / pathname / -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:

0	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 0 (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 { -AS alias | -AT destination | -CANCEL [PRT]x1 [,...xn] | -LIST | pathname }

If **pathname** is specified, causes the line printer to type out a specified file. PRIMOS assigns the file a number in the form **PRTx**, where x is a number between 001 and 200. (This number may vary on a per-installation basis.) 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 **-AS** option spools the file under an **alias**, or different filename. The **-AT** option sends the file to a line printer at the indicated destination. The **-CANCEL** option removes file or files identified by [PRTx] 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 | ME | NETWORK | UNITS | USERS }

Returns system status information indicated by specified options. **ALL** returns all status 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). The ME option returns user-related status information, including user-number, line number, open file units, etc.

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 program output to terminal to stop without returning to Primos command level. Programs can be halted by hitting CONTROL-S. 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.

USERS

Returns the number of users logged into PRIMOS at any given time.

13

BASIC/VM commands

The following is an alphabetized description of all BASIC/VM system commands. Commands are issued at BASIC/VM command level, in response to the BASIC system prompt character, ">". Commands may be typed in uppercase or lowercase letters. Some commands may also be used as statements and are so indicated. Command abbreviations are in rust. 18

ALTER line-number 18

Enters an editing mode to allow modification of indicated program 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	Function
A/**string**/	Append **string** to end of line.
B**nn**	Move pointer back **nn** characters (where nn is any integer).
C**c**	Copy line up to but not including **c** (where c is any character).
D**c**	Delete line up to but not including **c**.
E**n**	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.)
M**n**	Move **n** characters.
N	Reverse meaning of next C or D parameter (copy until character $\leqslant$ c, or delete until character $\geqslant$ 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 18

Attaches to directory specified by **pathname**; may have one of the following formats:

1. * > **sub-ufd-name**
   where * indicates the current directory
2. $\left\{\begin{matrix}< * > \\ < \textbf{disk} >\end{matrix}\right\}$ **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.
3. **ufd-name [> sub-ufd-name]**
   where directory named by **ufd-name** is located on the current disk.

```
ATTACH <6>MANUALS>REV16>PROGCOMP>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.

18| ▶ BREAK {ON / OFF} **lin-num-1 [,...lin-num-n]**

Sets and unsets breakpoints at indicated statement lines for debugging. **line-1** through **line-n** are statements at which the program is instructed to stop. A maximum of 10 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.

18| ▶ CATALOG **[option(s)]**

Lists all filenames under the current UFD, plus option information, if specified. **option(s)** are any or all of the following:

Option	Description
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.

18| ▶ 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.

19.0| ▶ COMINP { CONTINUE / **filename** / PAUSE / TTY }

19.0| Opens and reads commands in command file of specified **filename**. 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 14.

Note

As a command, COMINP takes an unquoted argument: as a statement, it takes a legal BASIC string argument.

18| ▶ 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 (such as, 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

18

Resumes program execution after a PAUSE or a breakpoint.

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

18

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]

18

If no **pathname** is specified, the currently compiled code in user memory is executed. When a binary file pathname is given, the binary file is immediately executed. When an uncompiled source file is specified, EXECUTE compiles and executes it. EXECUTE also displays 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, for example, a READ after a WRITE to a sequential file.

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

18

Deletes all except the specified lines. **lin-num-1** through **lin-num-n** are statement numbers to be saved. Statement numbers may be listed with comma separators, or they may be specified in a range, by using a dash. The statement numbers must be in ascending order.

FILE [pathname]

18

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:

- If a pathname is not specified, BASIC/VM automatically uses the name of the foreground file.
- 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:

```
FILE EXISTS.OK?
```

- All responses other than Y, YE, YES or OK, are interpreted as NO, and the following prompt appears:

```
NEW FILE NAME:
```

- A program need not be complete to be FILEd. It is advisable to FILE periodically 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.

LBPS

Lists currently set breakpoints. Breakpoints are set by the BREAK ON command.

18 ▶ **LENGTH**

Reports the number of statements in the current program.

18 ▶ **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 comma separators, or specified in a range, the beginning and end of which are separated by a dash.

18 ▶ **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.

18 ▶ **NEW [pathname]**

Indicates to BASIC/VM that a new foreground file is to be created with the specified name. If this **pathname** already exists, BASICV asks, FILE EXISTS. OK?. If the answer is "YES" or "OK", all lines previously in the foreground are overwritten.

18 ▶ **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
```

▶ **PERF** { **ON** | **OFF** | **TABLE** | **HIST** } { **[lin-num-1 [-lin-num-2]]** | **[screen-size]** { **CNT** | **AVG** | **TTL** } **[lin-num-1 [-lin-num-2]]** }

Measures program performance efficiency. The **ON** and **OFF** arguments turn performance measurement on or off, respectively. PERF ON must be issued prior to program compilation.

TABLE option prints the following performance information in tabular form: statement number (SN), number of times each statement was executed (CNT), the average statement execution time (AVG), the standard deviation of execution time (DEV), the total running time of each statement (TTL) and the total squared sum of the run-time of each statement (SQSUM). Times are measured in "ticks", typically 3.03 msec per tick. **lin-num-1** specifies the line number at which to begin displaying data. **lin-num-2** indicates the program line number at which to stop displaying performance data. Default is last line in program.

HIST displays data in histogram form, scaled according to **screen-size**; default size is currently set BASICV margin. (Default margin is 80 characters). The **CNT, AVG, TTL, lin-num-1** and **lin-num-2** options are identical to those described above. The symbols used to plot data in histogram displays are:

* (CNT)
. (AVG)
+ (TTL)

PURGE [pathname] 18

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 14.

QUIT 18

Returns control to PRIMOS from BASIC/VM command level. QUIT closes all files opened by BASIC/VM, and deletes any temporary files created by BASIC/VM.

RENAME newname 18

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] 18

Renumbers statements in the foreground program. new-start is the number which begins the new sequence. (Default: 100). old-start is the existing line number at which to begin renumbering. (Default: lowest numbered line). new-incr specifies increment value. (Default: 10).

RUN[NH] [lin-num] 18

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 } 18

Used to examine program logic or flow control. 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).

TYPE pathname 18

Displays the contents of the specified non-foreground file at the terminal, but does not replace the file currently in foreground.

14

BASIC/VM statements

The following is an alphabetized description of all BASIC/VM statements and their formats. Conventions are identical to those used for BASIC/VM system commands in Section 13. No abbreviations are accepted. Statements which can be used as commands are so indicated.

BASIC/VM CONVENTIONS

All PRIMOS command conventions, as listed in Section 2 or Section 12, also apply to BASIC/VM commands and statements. In addition, the following parameter representations are used throughout this section:

Parameter	Meaning
arg	Function argument
con	Constant (numeric or string)
(dim)	Dimension for array or matrix; a numeric item
expr	An expression; that is, 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** / KEY } = **str-expr-2 keylist**

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 secondary key(s). * indicates repetition of sequence as necessary.

CALL **subr_name (arg, [, arg] ...)**

Calls any declared and shared system, non-system or library routine from within a BASIC/VM program. For details, see Section 6 of this guide.

18

CHAIN **pathname**

Closes all open files and transfers program control to external program specified by **pathname**.

CHANGE **num-array** TO **str-var**

element A(0) contains the length of A$, or 4. A(1) contains the decimal code of W, which is 215, and so on. Conversely, if array A is changed to A$, the resulting string length is controlled by the value in A(0).

▶ **CLOSE #unit-1[,...unit-n]**

Closes files 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.

19.0| ▶ **COMINP str-expr** where **str-expr** is: { CONTINUE | filename | PAUSE | TTY }

19.0| Stops execution of current program and executes commands from command file specified by **filename**. 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 13.

▶ **DATA item-1[,...item-n]**

Lists numeric and string constants to be accessed 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 | APPEND] **FILE #unit = filename [,type-code] [,record-size]**

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 14-1, 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

Sets up a user-defined numeric or string function of one or more lines. The last line in the function definition must be FNEND. **var** is a simple numeric or string variable. **arg-1** to **arg-**

n are dummy arguments for the function; they may be numeric or string scalar variables. The defined function is not executed until referenced in the program, at which point control shifts to the function definition until FNEND is reached.

▶ 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-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**, (numeric expression in the range 1 to 1800), 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. | 19.0

▶ 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; always 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**, a numeric expression; and the final value of the index is represented by **end**, a numeric expression. When the index attains this value, loop execution stops. The STEP increment has a default value of 1. | 19.0

▶ FOR **index = start** [STEP **incr**] { WHILE | UNTIL } **condition-expr**

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 for loops with modifiers.

Table 14-1. File Type-Codes

Type-Code	Access Method	Contents
ASC (default)	SAM	ASCII data, formatted like terminal output with spaces as data delimiters. Commas, colons and semicolons define 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 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 10 at the start of each record. Can 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. Provides maximum precision and speed of access, 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	SEGDR	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	MIDAS	Multiple Index Data Access files. Created by Prime-supplied MIDAS utilities.

The following are examples of legal and illegal loop nesting.

Two-level Nesting (Legal):

```
FOR I1   UNTIL I1=13
  FOR I2 = 1 TO 13
      .
      .
      .
  NEXT I2
NEXT I1
```

Three-level Nesting (Legal):

```
FOR I1 = 1 TO 10
  FOR I2 = 1 TO 10
     FOR I3 = 1 TO 10
        .
        .
        .
     NEXT I3
  NEXT I2
NEXT I1
```

Examples of *unacceptable* nesting techniques are:

Two-level Nesting (Illegal):

```
FOR I1 UNTIL I1=13
  FOR I2 = 1 TO 10
     .
     .
     .
NEXT I1
  NEXT I2
```

Three-level Nesting (Illegal):

```
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 FOR, WHILE, UNTIL and UNLESS may be used with all executable BASIC statements. However, UNLESS may NOT be used in 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** | **THEN statement-1** } [**ELSE** { **statement-2** | **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** / THEN **statement** } ELSE { **line** / **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.

Strings entered in quotes will be accepted as input except for the quote delimiters which are dropped. This allows leading and trailing blanks to be included in string values.

Trailing commas and excess data are ignored. 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-1 [,...var-n] = expr

The assignment statement, used to assign values to numeric or string variables or array elements; the keyword LET is optional. **var-1 - var-n** represent numeric or string variables or array elements. Up to 100 variables may appear on the left side of the assignment statement. **expr** is a numeric value, string expression or another variable.

LOCAL { var-1 [...,var-n] / DIM var-1(dim-1)[,(dim-2)] }

Declares the listed variables or array names, **var-1** to **var-n**, to be local to the function definition in which they appear. **(dim-1)** and **(dim-2)** represent dimensions for a one- or two-dimensional array or matrix. Local variables and arrays maintain their values over many calls to a function and are not altered by program operations external to the function definition. Local variables and arrays are similar to function arguments in that they cannot be LISTed during a PAUSE or BREAK.

MARGIN {value / OFF}

Sets number of characters per line to **value**, a numeric expression. Range is 1 to 1000; the default is 80. MARGIN OFF turns off all margin settings. |19.0

MAT mat = {ZER / CON / IDN / NULL} [(dim-1) / (dim-1,dim-2)]

Sets initial value of matrix elements to zero, one, identity or null. 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 0; 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, for example, 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 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 the target matrix, mat-1. The resulting values in mat-1 can be multiplied by mat-2 to yeild 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 (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 force matrix elements into columns which are 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.

19.0| ### ON num-expr GOSUB lin-num-1,...lin-num-n [ELSE {GOTO / GOSUB} lin-num]

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
19.0| message will be displayed unless the ELSE GOTO or ELSE GOSUB clause is specified. Control is then transferred to the line number, **lin-num**, indicated.

ON num-expr GOTO lin-num-1,...lin-num-n [ELSE GOTO in-num]

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 in order for conditional transfer to occur. If the expression value is out-of-range, an error message is displayed unless an ELSE GOTO directs control to an alternate line (lin-num) in the program.

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. The ERR, ERL, and ERR$ (num-expr) functions are associated with the ON ERROR GOTO statement.

ERR	Function set to the code number of the error which activated the ON ERROR statement
ERL	Function set to line number being executed when the error occurred
ERR$(num-expr)	Function which outputs actual text of the error message associated with an error code represented by a numeric expression, num-expr

The ERROR OFF statement cancels 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**, for example, when an invalid number is entered.

ON QUIT GOTO lin-num

Sets up line number to which program control will be transferred when the user hits CTRL-P or the BREAK key during program execution. QUIT-trapping is turned off by the QUIT ERROR OFF statement.

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 { SEQ / [num-expr] = str-expr / 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 =0 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 / TAB / SPA } (num)] ,...item-n, [{ LIN / TAB / SPA } (num)]]

Prints formatted information at the terminal. **Item-1** to **item-n** represent numeric and/or string values.

19.0 **LIN** forces the specified number **(num)** of carriage return-line feed combinations between items in the output if num is greater than 0. If num, a numeric expression, is less than 0, it forces that many line feeds only: if num = 0, 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 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 14-2.

Table 14-2. 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.
-30	#####	30	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 (.)		
20	#####.##	20.00	Positions to right of decimal point are zero filled.
29.347	#####.##	29.35	Item is rounded off.
789012.344	#####.##	*****	If number is too large for the specified field asterisks are printed.
	COMMA FORMAT SPECIFICATIONS (,)		
30.6	+$,###.##	+$ 30.60	A space is substituted for comma when the leading digits are blank.
2000	#,###.	2,000.	Comma is printed in indicated position.
00033	++##,###	+00,033	Comma is printed when the leading zeros are not suppressed.

VERTICAL (UP ARROW) FORMAT SPECIFICATIONS (^)

170.35	+##.##^^^^	+17.03E+01
1.2	+##.##^^^^	+12.00E-01
6002.35	+###.##^^^^	+600.23E+01

Note

If more than four up arrows are used, the corresponding number of exponent digits will be printed.

PLUS SIGN FORMAT SPECIFICATIONS (+)

20.5	+##.##	+20.50	Plus sign printed where indicated; item is positive.
1.01	+##.##	+ 1.01	Leading zeros print as blanks.
-1.236	+##.##	- 1.24	Minus sign printed when item is negative.
-234.0	+##.##	*****	If number is too large for the specified field, asterisks are printed.

MINUS SIGN FORMAT SPECIFICATIONS (–)

20.5	###.##-	20.50	Sign discarded if item positive.
000.01	###.##-	.01	Leading zeros immediately to the left of the decimal point are suppressed. Minus sign not printed when item is positive.
-234.0	###.##-	234.00-	Sign printed as indicated.
-20	---.##	-20.00	Floating minus signs are treated as digit positions.
-200	---.##	******	Number does not agree with the format; asterisks are printed.
2	---.##	2.00	Item is positive; minus sign suppressed.

DOLLAR SIGN FORMAT SPECIFICATIONS ($)

30.512	$###.##	$ 30.51	Dollar sign printed.
-30.512	$###.##+	$ 30.51-	Negative item; minus sign printed where indicated.
13.20	+$$$$#.##	+ $13.20	Floating dollar sign printed immediately prior to leftmost significant digit.

Table 14-3. 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.

QUIT ERROR OFF

Turns off all QUIT-trapping set by ON QUIT GOTO statement. When the CTRL-P or BREAK key is hit subsequent to this statement, control returnto BASICV command level. (Does not cause control to return to PRIMOS.)

RANDOMIZE

Resets random number generator (RND function) at any point in a program. See Section 10, RND.

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 [SEQ / num-expr]=str-expr / 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** = 0, 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 =0 or is unspecified, pointer is positioned to primary key (default).

STOP

Causes termination of program execution. Returns message: STOP AT LINE lin-num.

SUB FORTRAN subr_name (arg-format [, arg-format] ...)

Declares any shared system, non-system or library routine which observes the FORTRAN calling sequence inside a BASIC/VM program. For details see Section 6 of this guide. 18

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.

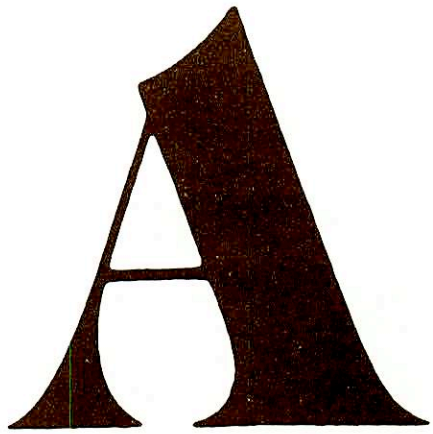
WRITE #unit, item-1 [,...item-n]

Writes data 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 a WRITE statement, all subsequent records in file are truncated.

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

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 14-2 and 14-3 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 cause items to be printed with no intervening characters or spaces. A format-string may be either a string constant or a string variable.



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 Program 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

>RUNNH
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$(V):A:'=':
590   ENTER U,M,C
600   IF C <> Q THEN DO
610     IF M >= 0 THEN DO
620       PRINT 'WRONG':N$:'.'
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$(V):A:'=':B*A
700   DOEND
710   ELSE DO
720     D = INT(3*RND(0) + 1)
730     IF D = 1 THEN PRINT 'RIGHT,':N$:'.'
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

>RUNNH
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?
!10
READY?
!YES
6 + 6 = 12
VERY GOOD.
YOU TOOK 2 SECONDS TO GET THE ANSWER.
MORE?
!YES
4 + 6 = 10
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 = 90
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 = Ø    ! N = # OF WORD DELIMITERS (SPACES)
190
200  FOR I2 = 1 TO L2    ! COUNT # OF WORDS
210       IF SUB(X$, I2) = ' ' THEN N = N + 1    ! IF SPACE, INCREMENT
220  NEXT I2
230
240  IF N = Ø THEN DO
250       A2$ = 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 <> Ø 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 'Simple 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.
>RUNNH
Simple 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
```

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
130		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
140		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
150		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
160		Space

Decimal value (with parity on)	ASCII character	Explanation
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	0	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	

Decimal value (with parity on)	ASCII character	Explanation
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

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 < = 0
50	SORT - ARGUMENT < 0
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
60	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 0
67	FATAL MIDAS ERROR
68	0 RAISED TO 0 OR A NEGATIVE POWER
69	CONSTANT ON LEFT SIDE OF ASSIGNMENT STATEMENT
70	MIDAS CONCURRENCY ERROR
71	(reserved)
72	UNKNOWN ARITHMETIC ERROR

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.

binary file: A translation of a source file generated by a language translator (PMA, COBOL, FTN, RPG). Such files are in the format required as input to the loaders. Also called "object file".

byte: 8 bits: 1 ASCII character.

condition mechanism: A PRIMOS facility which responds to conditions that would normally cause program termination. Rather than terminating the program immediately, the condition mechanism calls an on-unit to take some diagnostic or remedial action. A list of conditions handled by PRIMOS' condition mechanism is given in the Subroutine Reference Guide.

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 directories** in Section 2.

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 (CMDNC0). It is invoked by name, and executes in user address space. No system-wide abbreviations exist for external commands. Users may define abbreviations for external commands using the ABBREV command.

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, 0-9, – # $ – . * &

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
PH_filename	Phantom command 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 127 ('177) 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 is guaranteed at least 16 file units at a time. The maximum number of units that a user may have open simultaneously varies per installation; the default is 128. PRIMOS always reserves units 0 and 127 for its own use. In addition, certain commands or activities use particular unit numbers by default:

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. See the discussion on **Home vs current directories** in Section 2.

identity: The addressing mode plus its associated repertoire of computer instructions. Programs compiled in 32-R or 64-R 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. PRIMOS-defined abbreviations exist for external commands.

keys, file protection: Specify file protection, as in the PROTEC command.

0	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.

mode: An addressing scheme. The mode used determines the construction of the computer instructions by a compiler or assembler. (See **identity**.)

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**.

on-unit: A begin block (in PL/I) or subroutine (in FORTRAN, COBOL, or PL/I) which is called by the condition mechanism to handle error conditions. PRIMOS has on-units for all conditions it recognizes. Users may also define on-units within any procedure they write. User-written on-units take precedence over system ones.

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 1024 16-bit words within a segment (512 words on Prime 300).

partition: A portion *or all* of a multi-head 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 Section 2.

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.

procedure: In FORTRAN, a subroutine or function. In PL/I, any subroutine, function, or program. (In PL/I, procedures may contain other procedures.) In COBOL, the term usually refers to one or more related paragraphs or section within the Procedure Division. Procedures direct the computer to perform a particular operation or a series of operations.

process: A particular program running in a particular address space.

reserved characters: The following characters are reserved by PRIMOS for special uses. They may not be used as part of PRIMOS command lines.

{ } [ ] ! { } ; ^ " ? : ~ (delete or rubout key) also: | / + @ '

runfile: Executable version of a program, consisting of the loaded binary file, subroutines and library entries used by the program, COMMON areas, initial settings, etc. (Created using LOAD or SEG.)

SEG: Prime's segmented loading 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 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 logout. 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 you to halt terminal output by typing CONTROL-S. Output may be resumed by typing CONTROL-Q. Also sets terminal to full duplex mode (default).
-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 a 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, letter 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 automatically 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/**str1**/**str2**/	Replaces string 1(**str1**) with string 2(**str2**).
DELETE[n]	Deletes **n** lines including the current line.
DUNLOAD	Copies specified number 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 begining with given **string**.
INSERT **newline**	Inserts newline following current line. **newline** becomes current line.
LOAD **filename**	Copies contents of **filename** into work file under current line.
LOCATE **string**	Finds first line containing **string**.
NEXT [n]	Moves the pointer **n** lines. Positive or negative values accepted.
OVERLAY **string**	Overlays given **string** over current line, starting in column 1.
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 uppercase letters, as only uppercase 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 PRIMOS Commands Reference Guide.

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
INPUT A
INPUT B
PRINT A*B
H=A/B
IF H<Ø THEN 1Ø
PRINT H
END

OK, number
INTREENAME, OUTTREENAME, START, INCR,
ex ex3 1Ø 1Ø
OK, slist ex3
  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 accept commands and data input from a specified file rather than from the user's terminal. Command files are usually created with the EDITOR.

filename	The name of the file from which input is to be read.
funit	The PRIMOS file unit number on which the input file is to be opened. If omitted, File Unit 6 is assumed.

options Specifies control flow. -TTY tells PRIMOS to resume accepting input from the terminal. A command file should end with CO -TTY. The other options are detailed in Reference Guide, PRIMOS Commands.

Command input files are especially useful for repetitive processes such as displaying system information, deleting temporary files, 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.

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 an error message to be displayed.

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 files are useful when the user wants to keep a record of terminal transactions. PRIMOS opens file unit 127 and writes all command input and output responses to the file specified by filename.

Opening a COMO file: To open a COMOUT file, type COMO, followed by a filename:

```
OK, como record
```

This command line arranges for subsequent terminal output to be written to the file named RECORD. Commands are echoed, and responses continue to be displayed at the terminal. The file named RECORD is overwritten if it already exists.

Turning off terminal dislay: To inhibit echoing of any commands typed or responses displayed on the terminal screen, type:

```
OK, como -ntty
```

Terminal output continues to the file named RECORD, but nothing is displayed at the terminal.

Resuming terminal display: To resume display of terminal output, type:

```
OK, como -tty
```

Any commands typed at the terminal, as well as system responses, will now be displayed on the terminal screen. The output continues to be recorded in the COMO file RECORD.

Closing a COMO file: To close a COMO file, use the -E[ND] option:

```
OK, como -e
```

The file RECORD is now closed, and no further terminal output is written to the file. Note that command output files cannot be closed by CLOSE ALL; they must be closed explicitly by COMO -END, CLOSE unit -number or CLOSE filename. For example, to close RECORD, you could also issue one of these commands:

```
OK, close record
OK, close 127
```

Use the STAT UN command to obtain a list of open file units if you are not sure which ones are currently open.

FILE UTILITY (FUTIL)

FUTIL is a file utility command for copying, deleting, and listing files and directories. FUTIL is most often used for copying files and directories from one directory to another. It is also useful for deleting groups of files, entire directories, segment directories, and MIDAS files.

FUTIL accepts many subcommands, all of which are listed in the Reference Guide, PRIMOS Commands, or the PRIMOS Programmer's Companion. Only the ones most immediately useful to BASIC programmers are described in this Appendix.

Invoking FUTIL

To invoke FUTIL, type FUTIL. When ready, FUTIL prints the prompt character >, and waits for a command string from the user terminal. FUTIL accepts either upper or lowercase input, except that passwords must be entered *exactly* as they have been created. (Most other commands will convert passwords to uppercase before attempting the match. FUTIL does not.) To abort long operations, such as LISTF, type BREAK; restart FUTIL by typing S 1000.

To use FUTIL, type one of the FUTIL subcommands listed below, followed by a carriage return, and wait for the prompt character before issuing the next command. The erase (") and kill (?) characters are supported in both command and subcommand lines.

Copying files and directories

FUTIL provides several commands which allow the user to copy files, directories, or directory trees from one location to another. These commands, their functions and and formats are listed below:

COPY	Copies files (as many as will fit on line).
TRECPY	Copies directory trees.
UFDCPY	Copies entire UFD structure (complete with all files).
TO	Specifies directory to which file(s) or directories are to be copied: accepts a pathname. Default is home directory.
FROM	Directory from which files or directories are to be copied. Accepts a pathname. Default is home directory.

The general formats of these comands are:

COPY	pathname [new-name],[pathname new-name]
TRECPY	pathname
UFDCPY	

To move files and/or directories from one directory to another, the following general steps are taken:

1. Invoke FUTIL.
2. Define the FROM directory, unless it is the current directory.
3. Define the TO directory, unless it is the current directory.
4. Define the files, segdirs, etc., to be copied.
5. Indicate new filenames for copied files (optional).

Suppose we want to copy several files from another directory to the current working directory. The pathname of the files to be copied must be specified. To change the name of any file being copied, simply specify the old-name, then the new-name. Use a comma to separate filenames or pairs of filenames being copied. For example:

```
OK, FUTIL
[FUTIL rev 17.0]
>FROM <1>MARINE>NAUTILUS
>COPY HITS, MISSES ZEROES
>QUIT
OK,
```

The files HITS and ZEROES (formerly MISSES) are copied to the current directory. The file MISSES is renamed ZEROES in the current directory, but its name is not changed in the original, or FROM, directory. Notice that a TO-directory was not specified. If the TO-directory is not explicitly indicated, FUTIL assumes it to be the current directory. The subcommand QUIT, abbreviated "Q", returns the user to PRIMOS command level.

Deleting files and directories

The commands for deleting files, segment directories, MIDAS files, directory trees and UFDs are:

DELETE	Deletes specified files from FROM directory.
TREDEL	Deletes specified directory trees or segment directories, including MIDAS files, from FROM directory.
UFDDEL	Deletes entire current UFD and everything in it.

The user must have read, write, delete/truncate access rights to delete any file. The same general steps outlined above for file copying are followed in file and directory deletion.

The following examples show how segment directories and MIDAS files are deleted with FUTIL.

```
OK, futil
[FUTIL rev 17.0]
>from <*>tekman>progs
>tredel sega
> q
```

In the example above, the segment directory, SEGA, is deleted with the TREDEL option of FUTIL. Notice that a pathname is given, indicating that the segment directory is located under the current directory, (which is represented by the ("* > " symbol), in a sub-UFD called PROGS. If the segment directory is located in the current directory, no FROM-directory specification is necessary.

A MIDAS file can be deleted in the same manner as a segment directory. For example, if MIDAS.A is a MIDAS file in the current directory, the following dialog represents the deletion process:

```
OK, futil
[FUTIL rev 17.Ø]
>tredel midas.a
>q
OK,
```

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 topics covered are: access methods and file properties, altering files, and truncation patterns in each file type. This information may help programmers decide which file types to use in various data handling situations.

DATA STORAGE PATTERNS

Each type of file stores data differently. The storage patterns of ASCII and binary files are described below. Those of SEGDIR and MIDAS files are beyond the scope of this book. Refer to the Subroutine Reference Guide and the MIDAS Reference Guide, available under separate cover.

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. There are two major data storage methods: ASCII and Binary.

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, such as, decimal points, dollar signs, and so forth.

The main feature of ASCII files is their ease of inspection. They can be LISTed, 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 just as they are in ASCII files, that is, in ASCII code. Numeric data are stored in internal machine format, that is, in 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 by the user through LIST, TYPE or ED (the PRIMOS editor).

Intra-record data storage

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 in Section 8. These descriptions are intended to aid you in selecting the files best suited to your particular data storage and access requirements.

ASC files: ASCII files are the default file type in BASIC/VM. They store data exactly like terminal output. Semicolons, 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 item by the number of spaces indicated by the 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: 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:

```
12,13,14,15
```

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 value '$12,000' is written to an ASCSEP file it is stored as:

```
$12,000,
```

When read back, the following occurs:

```
>READ#1, A$
>PRINT A$
$12
```

The commas are interpreted as delimiters in this case. This problem can be remedied by using an alternate form of the READ statement, READLINE. READLINE accepts the entire contents of one ASCII record, including commas, semicolons, colons and spaces, as one datum.

ASCSEP files are accessed sequentially, 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 each record is marked by a carriage return (CR). The delimiters which influence 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: 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, unless a previously-created ASCLN file is re-opened without file-type specification. In this case, the line numbers are returned during file READs.

The LIN#(unit) function (see Section 10) returns the inserted line number of the current record. In other words, if the pointer is at the second file record, the LIN#(unit) function prints the record number as 20. For example:

```
100 DEFINE FILE #1 = 'LN', ASCLN
110 PRINT 'TOP OF FILE:': LIN#(1)
120 WRITE #1, 'FIRST RECORD'
130 WRITE #1, 'SECOND RECORD'
140 REWIND #1
150 READ #1, A$
160 PRINT LIN#(1):A$
170 READ #1, B$
180 PRINT LIN#(1):B$
190 PRINT 'NOW AT END OF FILE:':LIN#(1)
200 CLOSE #1
210 END
>RUNNH
TOP OF FILE: 0
10 FIRST RECORD
20 SECOND RECORD
NOW AT END OF FILE: 20
>TYPE LN
   10 FIRST RECORD,
   20 SECOND RECORD,
```

The first record in the file which contains data has a line number of 10. The second record has a line number of 20. The top of the file is record 0.

ASCLN files are the only data files which can 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 storing data that must be frequently updated or 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. Thus, all records in a direct access file are of equal length. 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 are usually output at the terminal due to blank-padding. 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 current location of a record pointer in a direct access file. The LIN#(unit) function can be used to determine the actual record number at which the current I/O operation is being performed. LIN#(unit) returns the first record in the file as 0. The first record in the file can be positioned to by the statement: POSITION #unit TO 1. However, the LIN# function returns this as record 0.

```
100 DEFINE FILE #2 = 'LNDA', ASCDA
120 WRITE #2, 'JUPITER'
130 WRITE #2, 'MARS'
140 REWIND #2
150 PRINT 'TOP OF FILE:':LIN#(2)
160 READ #2, A$
170 PRINT LIN#(2):A$
180 PRINT
190 READ #2, B$
200 PRINT LIN#(2):B$
210 PRINT 'END OF FILE AT RECORD:':LIN#(2)
220 CLOSE #2
230 END
>RUNNH
TOP OF FILE: 0
1 JUPITER

2 MARS
END OF FILE AT RECORD: 2
>TYPE LNDA
JUPITER,

MARS,
```

BIN and BINDA files: Binary files of both types have fixed-length records. String data are stored as in ASCII files, that is, in ASCII code; numeric data are stored in internal machine format, that is, four-word floating-point representation, ensuring complete data accuracy. Program execution is also expedited because less translation time is required during numeric data access. Any portion of a record not filled with data is zeroed out rather than blank-filled.

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.

ACCESS METHODS

Data files can be accessed by one of four 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 and are covered in detail in the Subroutine Reference Guide, and in the MIDAS Reference Guide.

Sequential access method (SAM)

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, the pointer must skip through each record in the file. It is not possible to 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 one and the desired one 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 is to be read, the pointer 'calculates' that 160 characters must be skipped before the third record is reached. When 160 characters have been bypassed, the pointer is positioned to the beginning of the third record. The importance of fixed-length records in direct access is readily apparent.

ACCOMMODATING LARGE DATA ITEMS

Occasionally you may find it necessary to put more data in a record than permitted by the previously defined record size. Each type of file handles this space problem differently, as described below.

Altering record size in 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 accommodate this item.

```
>DEFINE FILE #1='T1',ASC,4
>WRITE #1,'HARRY G. MUDD'
>TYPE T1
HARRY G.
 MUDD
```

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: ASCII line numbered 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. Data items which exceed the set record length are treated in the same manner described earlier.

```
>define file #1 = 'test', ascln, 5
>write #1, 'lesley'
>close #1
>type test
   10 les,
```

ASCSEP files: ASCII separated files differ from ASC and ASCLN files both in structure and 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 accommodate 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 do not affect subsequent WRITE statements as they do in ASC files.

BIN files: Binary files maintain data quite differently from ASCII files. Although the exact nature of intra-record data storage cannot be easily determined, 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
>REWIND #1
>READ #1, A
>PRINT A
12345678910.32
```

Altering record size in DAM files

The method used to retrieve data from a direct access file requires that all the records in the file 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 record size are written to a DAM file, one of two things can occur:

- If the current record is empty, and the datum exceeds the record size, the item is truncated.
- If a series of items, for example, A,B,C$, are included in a single WRITE statement, and C$ does 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 is truncated.

The following example shows what happens when large items are written to a direct access file with a record size limited to 16 characters:

```
10 DEFINE FILE #1 = 'TRZ', ASCDA,8
20 WRITE #1,1234,5678,'MAGNIFICENT'
30 WRITE #1, 'I AM EIGHTEEN CHAR', 123,456
40 REWIND #1
>RUNNH
>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 is added to the file. This causes a blank line to occur in the output when the file is TYPEd. For example:

```
10 DEFINE FILE #1 = 'BL1'
20 DEFINE FILE #2 = 'BL2', ASCSEP
```

```
30 WRITE #1,12
40 WRITE #2,12
50 WRITE #1
60 WRITE #2
70 WRITE #1,13
80 WRITE #2,13
>RUNNH
>TYPE BL1
12

13
>TYPE BL2
12,

13,
```

Truncating a file

Data written to a record that already contains data will overwrite existing data. If the file is CLOSEd immediately subsequent to the overwrite, the file will be truncated. Thus, the record just written to becomes the last record in the file. For 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

The impact of ASCII file properties on file READs is worth exploring in some detail. These READ features are applicable only to the default ASCII file type.

Reading default ASCII files

READ results vary with the data delimiters within each record, as explained earlier. The READ pointer interprets commas as "end-of-data" markers in all file types. Thus, it does not consider the ASC spacing delimiters "true" delimiters. ASC files are the only ones adversely affected by this feature.

If several numeric items are written to a file with various delimiters, as shown below:

```
>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
```

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, for example, 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. For example:

```
>DEFINE FILE #1 = 'ASCII', ASC
>WRITE #1, 'SUGAR', 12.ØØ
>WRITE #1, 'FLOUR', 5.ØØ: 'COFFEE'
>REWIND #1
>TYPE ASCII
SUGAR                 12
FLOUR                 5 COFFEE
>READ #1,A
INPUT DATA ERROR AT LINE  Ø

>READ #1,A$
>PRINT A$
FLOUR                 5 COFFEE
>REWIND #1
>READ #1,A$,B
INPUT DATA ERROR AT LINE  Ø

>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 cannot be read into the numeric variable specified.

Reading 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 is truncated at the position where the comma occurs.

- If the record contains both string and numeric data, "READ #unit, A", returns a numeric item only if it appears first in the record: "READ #unit, A$" returns either numeric or string values, depending on which occurs first in the record.
- String data can only be read into string variables, for instance, "READ #1, A$." Numeric data can be read into either numeric or string variables, for example, the expressions, "READ #1, A" or "READ #1,A$", both return a numeric value.

The following example deals with numeric and string READs done on an ASCSEP file:

```
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
```

Summary of READs on sequential files

The following example shows how the properties of sequential files influence the results of simple file READs. Remember that reading numeric values from a binary file into a string variable produces strange results. For this reason, binary files have been excluded from this example.

```
10 ON ERROR #1 GOTO 260
20 DEFINE FILE #1 = 'ASC'
30 DEFINE FILE  #2 = 'ASCSEP', ASCSEP
40 DEFINE FILE #3 = 'ASCLN', ASCLN
50 READ A,B,C$
60 DATA 123.45,48,'$100,000'
70 FOR I = 1 TO 3
80 WRITE #I, A,B,C$
90 REWIND #I
100 NEXT I
110 FOR N = 1 TO 3
120 PRINT 'NUMERIC READ FOR FILE ON UNIT #': N
130 PRINT
140 READ #N, A
150 PRINT A
160 REWIND #N
170 PRINT 'STRING READ FOR FILE ON UNIT #': N
180 PRINT
190 READ #N, A$
200 PRINT A$
210 PRINT
220 REWIND #N
230 NEXT N
240 CLOSE #1,2,3
```

```
250 GOTO 290
260 PRINT 'ERROR ON UNIT:':N
270 PRINT ERR$(ERR)
280 GOTO 160
290 END
>RUNNH
>TYPE ASC
123.45                48                      $100,000
>TYPE ASCSEP

123.45,48,$100,000,
>TYPE ASCLN
   10 123.45,48,$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
```

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 in Section 8.

READ* vs. READ

In default ASCII files, the READ* statement is most useful when records do not contain both numeric and string values. The variables for READing must be carefully chosen to avoid INPUT DATA ERROR messages. This requires some familiarity with the data being READ or otherwise manipulated.

Reading numeric values: The example below shows the difference between READ and READ* statements when used to retrieve numeric values from various sequential files.

```
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,
```

Reading string values: The following example DEFINEs and WRITEs string data to several sequential files. Both types of READs, READ and READ*, are done, for comparison, on various SAM file types.

```
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*
   10 RED,WHITE,
   20 BLUE,PURPLE,
```

F

Loading non-system library routines

LOADING NON-SYSTEM LIBRARY ROUTINES WITH BASIC/VM

Non-system library FORTRAN routines can be loaded with BASIC/VM. In the interest of system security, we recommend that only your System Administrator or supervisor have write access – or authorize write access to others – for using the files and directories that load the routines. There are circumstances where security at your site would be very important to the operation of the system; for example, in a school environment. For more information regarding the role and duties of the System Administrator, see the **System Administrator's Guide**.

Load non-system library routines with the following six-step procedure:

1. Include the routine's source file in sub-UFD BASICVSRC>SOURCE.
2. Edit BASICVSRC>SOURCE>FTNINT.USER.FTN to include the routine name.
3. Edit BASICVSRC>BASICV.BUILD.CPL to include loading of routine.
4. Run BASICVSRC>BASICV.BUILD.CPL to load BASICV.
5. Run BASICV>BASICV.INSTALL.COMI to install BASICV.
6. Run BASICV>BASICV.SHARE.COMI to share BASICV.

Note

If you are loading subroutines from the system library VAPPLB, Steps 1 and 3 are not necessary.

Example

A non-system library (user-written) FORTRAN routine named XYZ is to be called by a BASIC/VM program. The routine XYZ looks like this:

```
C     XYZ.FTN - WRITTEN BY USER 'JD' (JOHN DOE) - 3/28/82
C     THIS ROUTINE SHOULD BE LOADED WITH THE BASIC COMPILER
C     TO ALLOW ANYONE TO USE IT.
C
C XYZ FORTRAN SUBROUTINE
C
C  ROUTINE PRINTS VALUE OF ARGUMENT AND MODIFIES ARGUMENT BY TRIPLING IT
C
      SUBROUTINE XYZ(A)
C
      INTEGER*2 A
      WRITE(1,100) A
100   FORMAT('VALUE OF ARGUMENT PASSED :  ',F15.5)
C
      A=A*3
C
      RETURN
      END
```

The sample routine XYZ is to be declared and called from a BASIC/VM program that looks like this:

```
10 ! THE FOLLOWING BASIC/VM PROGRAM SERVES AS A SAMPLE SESSION
20 ! TO DEMONSTRATE FTN CALL FUNCTIONALITY.
30 !
40 ! THE SECOND LINE IS OUTPUT BY THE FORTRAN USER-DEFINED SUBROUTINE
50 ! 'XYZ' AND 'XYZ' REQUIRES AN ARGUMENT OF TYPE INTEGER*2.  IT RETURNS
60 ! AN ARGUMENT THREE TIMES ITS ORIGINAL VALUE.
70 !
80 SUB FORTRAN XYZ (INT)
90 A = 123
100 PRINT 'VALUE OF ARGUMENT TO BE PASSED :   ';A
110 CALL XYZ (A)
120 PRINT 'VALUE OF ARGUMENT RETURNED :   ';A
130 END
```

Follow the six-step procedure. Under most circumstances, it is better not to do Steps 1 through 5 from the supervisor terminal because these steps are very time-consuming – especially Steps 4 and 5 – and they can tie up other operations performed from the supervisor terminal. However, you must have write access to the files and directories, no matter what terminal you use. Step 6 can only be done from the supervisor terminal.

Using the sample routine and BASIC/VM program:

1. Include the routine's source file as XYZ.FTN in the sub-UFD BASICVSRC>SOURCE.

2. In BASICVSRC>SOURCE>FTNINT.USER.FTN, after the statement EXTERNAL SAMPLE, insert the statement EXTERNAL XYZ. Also, after the statement:

   IF (NAMEQ$ (NAME,LEN,'SAMPLE',6)) FTNINT = LOC(SAMPLE)

insert the statement:

```
IF (NAMEQ$ (NAME,LEN,'XYZ',3)) FTNINT = LOC(XYZ)
```

Your editing session for Step 2 would look something like this:

```
ed ftnint.user.ftn
EDIT
locate EXTERNAL SAMPLE
      EXTERNAL SAMPLE  /* NOTE THAT 'SAMPLE' IS SAMPLE ROUTINE NAME*/
insert       EXTERNAL XYZ
locate NAMEQ
      IF (NAMEQ$(NAME,LEN,'SAMPLE',6)) FTNINT = LOC(SAMPLE)
insert       IF (NAMEQ$(NAME,LEN,'XYZ',3)) FTNINT = LOC(XYZ)
file
```

Note that the fourth argument in the function (3) is the length of the subroutine's name in bytes (characters).

3. In BASICVSRC>BASICV.BUILD.CPL, after the line:

   ```
   FTN SAMPLE -64V -XREFS -SPO -DCLVAR -PBECB -LIST NO
   ```

   insert the line:

   ```
   FTN XYZ -64V -XREFS -SPO -DCLVAR -PBECB -LIST NO
   ```

   Your editing session for Step 3 would look something like this:

```
ed basicv.build.cpl
EDIT
locate SAMPLE
FTN SAMPLE -64V -XREFS -SPO -DCLVAR -PBECB -LIST NO
insert FTN XYZ -64V -XREFS -SPO -DCLVAR -PBECB -LIST NO
file
```

   FORTRAN compile options can be varied.

4. Run BASICVSRC>BASICV.BUILD.CPL with the RESUME command.

5. Run BASICV>BASICV.INSTALL.COMI with the COMINPUT command.

6. Run BASICV>BASICV.SHARE.COMI with the COMINPUT command. (This step can only be done from the supervisor terminal.)

Here is the output of the sample BASIC/VM program, which calls the sample routine XYZ:

```
>RUNNH
VALUE OF ARGUMENT TO BE PASSED :   123
VALUE OF ARGUMENT PASSED :         123.00000
VALUE OF ARGUMENT RETURNED :   369
```

A

B

C

D

E

F

N

O

P

Q

R

S

T

U

V

W

X,Y,Z