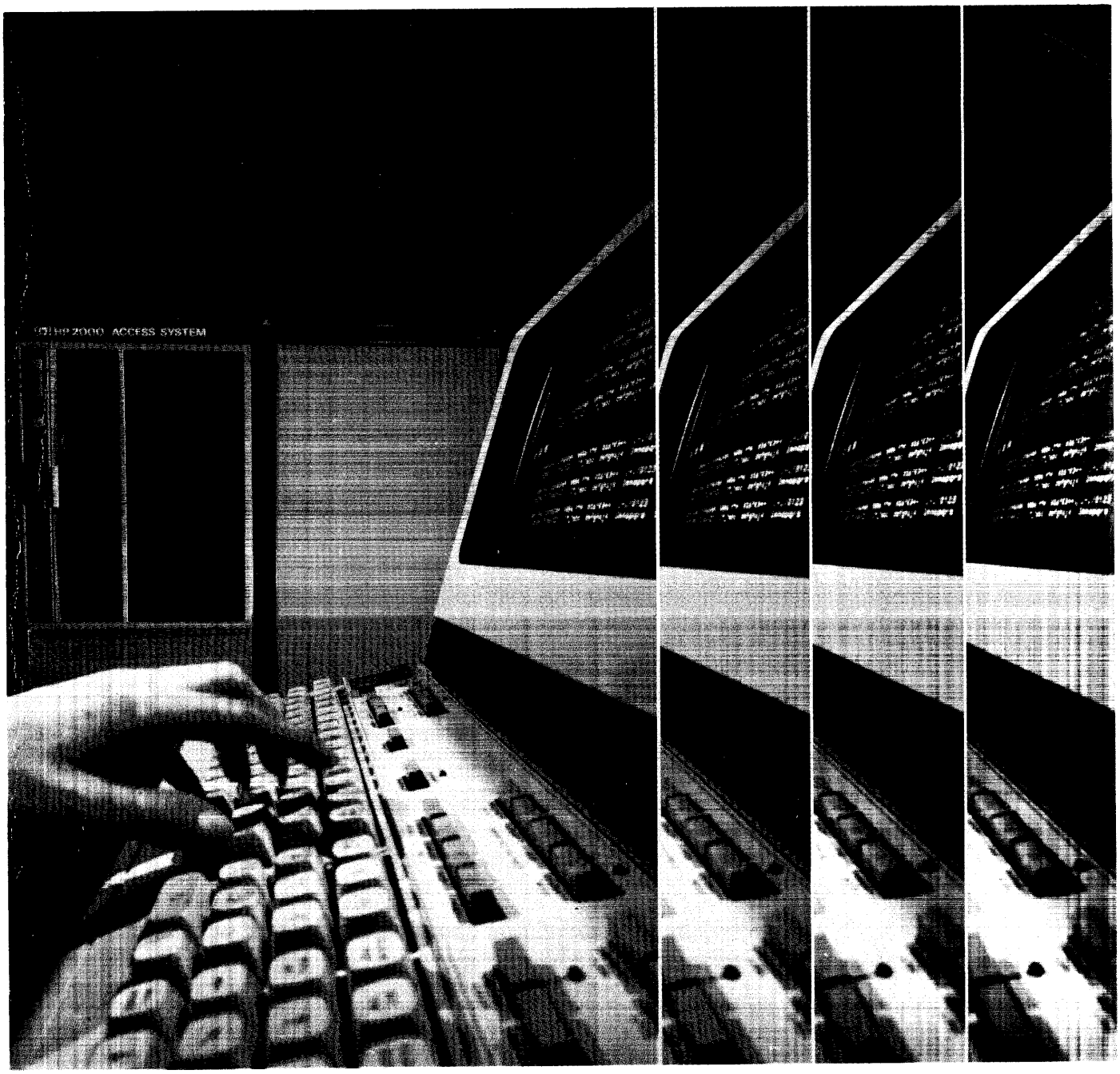
HEWLETT PACKARD

HP 2000/Access BASIC

Reference Manual



HP 2000/Access BASIC

Reference Manual

HEWLETT-PACKARD COMPANY
11000 WOLFE ROAD, CUPERTINO, CALIFORNIA, 95014

PART NO. 22687-90001
PRODUCT NO. 22687A

Printed in U.S.A. 9/75

LIST OF EFFECTIVE PAGES

Pages	Effective Date
Title	Sep 1975
ii to xi	Sep 1975
1-0 to 1-11	Sep 1975
2-1 to 2-26	Sep 1975
3-1 to 3-17	Sep 1975
4-1 to 4-11	Sep 1975
5-1 to 5-17	Sep 1975
6-1 to 6-9	Sep 1975
7-1 to 7-4	Sep 1975
8-1 to 8-7	Sep 1975
9-1 to 9-17	Sep 1975
10-1 to 10-29	Sep 1975
11-1 to 11-92	Sep 1975
A-1 to A-4	Sep 1975
B-1	Sep 1975
C-1 to C-7	Sep 1975
D-1 to D-3	Sep 1975
E-1 to E-2	Sep 1975
F-1 to F-10	Sep 1975

ACKNOWLEDGEMENT

Hewlett-Packard wishes to acknowledge the substantial contribution to the development of the 2000 Access System made by members of the professional staff of the University Computer Center of the University of Iowa. HP feels that this has been an unusually productive and cordial relationship between an industrial firm and an institution of higher education.

PREFACE

This publication is the user's reference manual for the HP 2000 Access System. It provides information for logging on and developing and executing BASIC language programs. Included are instructions for using the system's Remote Job Entry facility. Other manuals which may provide useful information when using this manual are:

- *Learning Timeshare BASIC* (22687-90009) — An introduction to the BASIC language and a tutorial explanation of statements and commands.
- *HP 2000 Access Operator's Manual* (22687-90005) — A guide to operating the 2000 Access System on a daily basis including administrative procedures.
- Pocket Guides (22687-90003, 22687-90007) — Summaries of system capabilities for user's and the system operator.
- *TSP/2000-HASP User's Manual* (20240-90002) — A guide to using the Telecommunications Supervisory applications Package (TSP) with the Remote Job Entry facility.
- *TSP/2000-HASP Application Manager's Manual* (20240-90001) — A manger's guide to administering the Telecommunications Supervisory applications Package (TSP).

This manual is organized into eleven sections, six appendices, and an index.

- Section I — Introducing 2000/Access BASIC. This section is a description of the system, its capabilities, and how to use it.
- Section II — Introduction to BASIC Programming. This section describes key elements of programming in BASIC on the 2000 Access System.
- Section III — Programming With Arrays. This section describes statements and techniques for using arrays in programs.
- Section IV — Programming With Strings. This section describes statements and techniques for using strings in programs.
- Section V — Files. This section describes the types of files used on the system, file read and write operation, and techniques of file organization.
- Section VI — Formatted Output. This section describes techniques of creating formatted output for applications such as report generation.
- Section VII — System Facilities. This section describes techniques for linking programs and passing data from one program to another.
- Section VIII — Security and the Library Hierarchy. This section describes the levels of program and file security available on the system. Techniques for using the security and library structure are discussed.

- Section IX — Using the Remote Job Entry Facility. This section describes the Remote Job Entry capability of the 2000 Access System and provides instructions for its use.

- Section X — Commands. This section provides the syntax and definition of all the user commands available.

- Section XI — BASIC Language Reference. This section provides a definition of the BASIC language. A rigorous definition of the terminology precedes the syntax and definition of the BASIC statements. Experienced programmers can use this section as a detailed reference for the BASIC language as implemented on the 2000 Access system. Less experienced programmers should first read the introductory sections of the manual, and if you are a beginning programmer, *Learning Timeshare BASIC* is recommended.

- Appendix A — Using the ASCII Character Set. This appendix provides a description of the ASCII character set and its use in BASIC language statements.

- Appendix B — How to Prepare A Paper Tape Off-Line. This appendix describes the preparation of paper tapes for input to the system.

- Appendix C — Error Messages. This appendix lists the error messages that you could receive when using the system.

- Appendix D — Terminal Interface. This appendix provides some of the operating characteristics of terminals used with the system.

- Appendix E — Additional Library Features. This appendix describes some of the special functions available through the system operator.

- Appendix F — Formal Syntax for 2000/Access BASIC. This appendix presents the BASIC language syntax in Backus Naur form (BNF).

The text in this manual is primarily for reference and has been written as a definition rather than an explanation of the BASIC language. Experienced programmers can use the detailed reference material given in Sections X and XI to look up the syntax of commands and statements. Less experienced programmers should read the entire manual.

Simple programming examples are used throughout the manual. The examples have been selected to demonstrate how the language is used and are not intended to be examples of efficient programming

CONTENTS

CONTENTS (continued)

CONTENTS (continued)

CONTENTS (continued)

CONTENTS (continued)

ILLUSTRATIONS

Figure **Page**

TABLES

Table **Page**

Figure 1-1. Typical HP 2000 Access System

INTRODUCING 2000/ACCESS BASIC

SECTION I

WHAT IS 2000 ACCESS?

2000 Access is a terminal-oriented computer system using a powerful version of the BASIC language. Up to 32 users may access the system simultaneously through hardwired terminals or over ordinary telephone lines using modems. The system may also use a variety of peripherals such as card readers, paper tape punches, magnetic tape drives, paper tape readers and line printers. A typical system is shown in figure 1-1.

In addition to system users there is a system manager and a system operator. The system manager is responsible for establishing the initial system configuration and setting operating policies. The system operator is responsible for daily system operation and the granting of system resources such as storage space and terminal time. In the remainder of this manual there will be additional references to the system manager and system operator functions. Detailed descriptions of their duties are contained in 2000 Access System Operator's Manual (22687-90005).

The remainder of this section describes briefly major system components, capabilities, and how to use the system. This manual assumes that you have programming experience in a language similar to BASIC and are familiar with the type of terminal used on your system.

The system is made up of three major components, the BASIC language, the operating system software, and the system hardware. The components allow you to make use of a variety of system resources.

BASIC LANGUAGE

Hewlett-Packard 2000/Access BASIC contains additional programming features beyond those in most versions of the BASIC language. These features, in the form of an extended set of statements, give you powerful tools for applications concerned with on-line data management and computation.

OPERATING SYSTEM SOFTWARE

The operating system software controls the overall system operation, supervises file activities, controls all input/output operations, and provides utility functions. System operation is discussed in detail in the HP 2000 Access System Operator's Manual (22687-90005).

SYSTEM HARDWARE

The system uses two computers, one for handling input/output operations (I/O processor) and another for executing BASIC programs (main processor). In addition it uses high speed disc drives for program and file storage, magnetic tape units for system backup and offline storage, and various terminals, printers, card readers, and other input/output devices. The peripheral devices available will vary depending on your system configuration. The organization of a

typical system is shown in figure 1-2. Information on individual system hardware units is contained in the installation, maintenance, and operator manuals for the specific device.

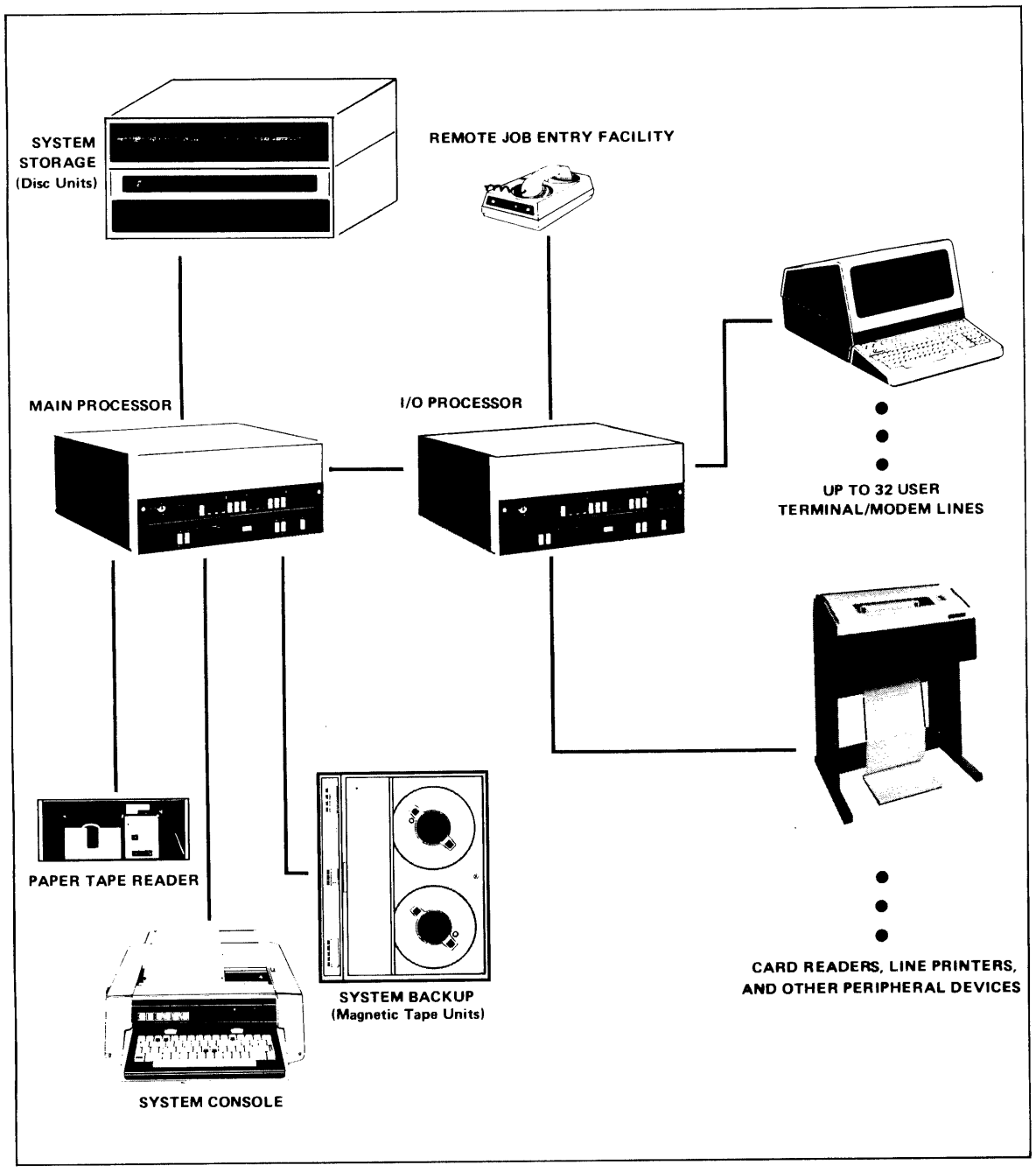


Figure 1-2. System Block Diagram

SYSTEM RESOURCES

The system resources consist of input/output devices, file supervision, terminal time, security, and remote job entry capabilities. These resources are available either directly through your terminal or on request from the system operator. The system operator accesses the system through the system console and performs privileged operations affecting system resources.

INPUT/OUTPUT DEVICES. Your terminal is the best example of a system input/output device. In addition to your terminal, a variety of input/output units are available for use as "non-sharable devices". These devices are peripherals such as line printers and magnetic tape drives. They are non-sharable in the sense that once you begin using a device it remains under your control and may not be used by others. The device is released by program or command termination. They may also be assigned for exclusive use from time to time by the system operator to a specific user. Once assigned to you, a line printer, for example, may not be used by other users. Devices are deassigned by the system operator and may be immediately reassigned to another user.

FILE SUPERVISION. The system controls access to files. You are given a limit to the amount of file space you may use at the time your account is opened by the system operator. Up to 65,535 records of file storage can be allocated to any one user account. This amount may be less depending on your system's configuration and operational policies. The amount of file space can be increased or decreased at any time by the system operator. This allows you to begin programming with a relatively small file space and increase it later as your storage needs grow.

SECURITY. When your account is opened by the system operator you will be assigned an account number and a security code (password). This password is used to limit the access to your account. Several people may share an account; or accounts may be assigned one to a user. Additional information on user accounts is contained in the description of the system account structure. Each program and file on the system may be assigned one of four levels of security.

REMOTE JOB ENTRY (RJE). The system may be used to provide a remote work station for a larger computer system concurrent with normal system operation. The RJE facilities are available directly through peripheral devices or from your program. You can then enter jobs, programs, or data in any language (FORTRAN, ALGOL, COBOL, etc.) available on the second system. A complete discussion of the RJE capabilities of the system and operating procedures is contained in Section IX of this manual.

TERMINAL TIME. When your account is opened by the system operator you are given a limit to the amount of terminal time you may use. Time up to a maximum of 65,535 minutes can be given. The system logs all terminal time used against the appropriate account. The time accrued may be reset to 0 by the system operator.

WHAT DOES THE SYSTEM DO?

The primary function of the system is to allow up to 32 concurrent users to develop and execute programs in the BASIC language. In addition the system provides you with resources for input/output, file maintenance, operator utility functions, system, program, and file security, and Remote Job Entry (RJE).

The BASIC language used on the system consists of statements for writing programs and commands for controlling both program execution and input/output operations. This manual assumes that you are familiar with a language similar to BASIC and that you know how to use the terminals available on your system.

The sample program in figure 1-3 illustrates some of the statements and commands available on the system for program access and execution. Program statements are numbered, commands are not. Special terminal keys are circled, and system responses are shaded.

Figure 1-3 illustrates logging on from a terminal, creating a data file, accessing, listing, and running a program. (The program is assumed to have been previously written and stored in your account library.) The program outputs a message, inputs string data, and stores the data on the file. A simple test for a "/" is made to terminate the program. A description of key program statements is provided.

OPERATION/STATEMENT	DESCRIPTION
(return) (linefeed)	
PLEASE LOG IN	System log in procedure
HELLO-B105,PASWRD (return)	User enters account number and password
(System Message)	Message varies with system
CREATE-FIL1,10 (return)	Create empty file FIL1 with 10 records
GET-PROG1 (return)	Get PROG1 from your library
LIST (return)	List program
PROG1	
10 REM BUILDS A LIST OF PROGRAM NAMES	Remark (not executed)
20 DIM A$(80)	Dimension a string
30 FILES FIL1	Specify file to be used
40 PRINT "TYPE / TO EXIT"	Output prompt messages
50 PRINT "ENTER PROGRAM NAME"	
60 INPUT A$	Accept input data
70 IF A$="/" THEN 100	Test for exit and branch to END
80 PRINT #1; A$	Write data sequentially to file
90 GOTO 50	Branch to statement 50
100 END	End of program
RUN (return)	Execute program
PROG1	
TYPE / TO EXIT	
ENTER PROGRAM NAME	
?TEST1 (return)	
ENTER PROGRAM NAME	
?TEST2 (return)	
ENTER PROGRAM NAME	
?/ (return)	
DONE	
BYE (return)	
System Message	

Figure 1-3. Example of Program Access and Execution

HOW DO YOU USE THE SYSTEM?

The 2000 Access system is designed to be extremely easy to use. It requires no complicated job control language; program entry is free field; and there is no separate compile operation. Using the system consists of obtaining an account number and password from the system operator, logging on, entering or retrieving a program, and running the program. In addition you can send messages to the computer operator requesting a number of auxiliary operations.

ACCOUNT AND LIBRARY SYSTEM

The accounts used on the system are made up of three types, system, group, and user. Each type of account has slightly different capabilities and performs a different function in the typical system. Each account has its own library for storage of programs and files. The following paragraphs discuss briefly each of the account types and their capabilities.

SYSTEM MASTER ACCOUNT. The system master account is numbered A000 and is used only by the system master or system operator. This account is used to hold the system library containing programs and files that can normally be accessed by all system users. A listing of the accessible contents of the system library can be obtained using the LIBRARY command.

The A000 account possesses additional capabilities. The A000 user has the ability to execute some system operator commands such as DIRECTORY, DUMP, and REPORT. A complete discussion of system master capabilities is given in the HP 2000 Access System Operator's Manual (22687-90005).

GROUP MASTER ACCOUNTS. The group master or group librarian account has an account number made up of a letter and a digit followed by two zeros. For example, A000 (System Master), Z900, J100, and K700 are all group master accounts. The group account contains programs and files which may be accessed by any account within the group. For example, C100 is a group master account and all accounts from C100 through C199 are members of the group. Figure 1-4 shows the account structure in a small system.

Group library programs and files can be accessed by members of the group in a similar manner to system library entries.

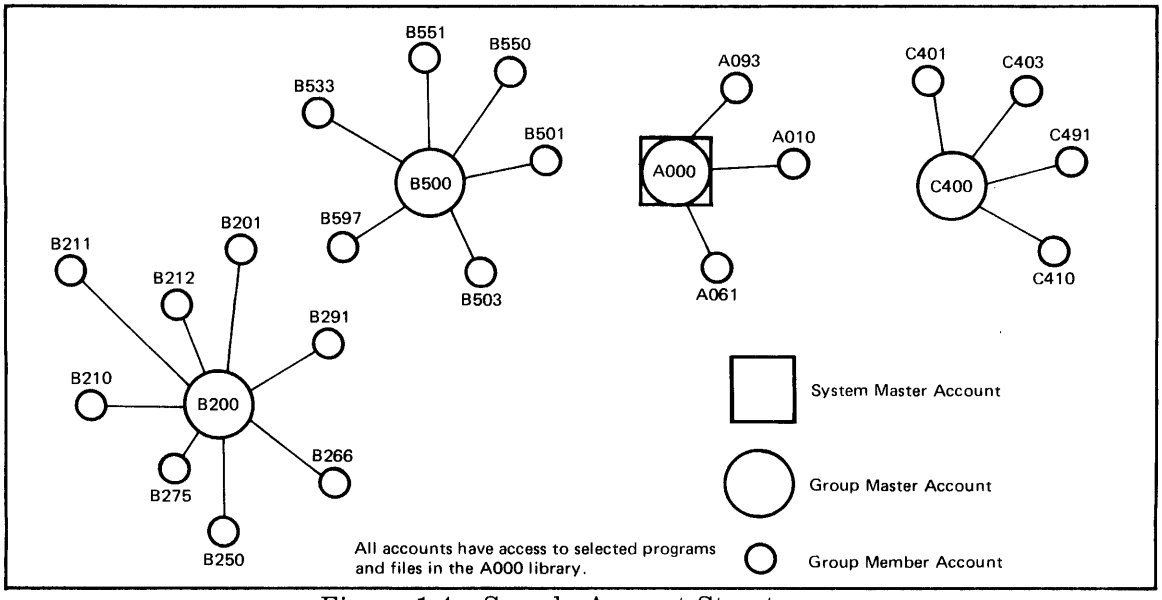


Figure 1-4. Sample Account Structure

INDIVIDUAL ACCOUNTS. The individual account has access to its own files and programs, its group library, and the system library. Note that the system library is the library belonging to account A000 and that each group library is the library belonging to the group master account. The individual account owner has control over his own library through the various commands and statements used to store, delete, or retrieve programs and files. In addition the owner may change the level of security for programs and files in his library.

OPERATING THE EQUIPMENT

You can use any of several terminal types with the system. The terminal used must generate characters using ASCII code. (An exception is the IBM 2741 terminal.) It is not necessary that the terminal be capable of generating the entire ASCII character set. If your terminal does not use the entire set, you may not be able to use the full capabilities of the system. For example, you can use all 128 ASCII characters but some terminals do not print the lower case alphabet. This is important in such applications as text editing and report generation, but is not necessary for some data processing or analytical applications.

CONNECTING TO THE SYSTEM. To log onto the system, connection must be established between your terminal and the system. The way that this connection is made varies depending on the type of terminal used. If your terminal is wired directly to the system (hardwired) all that is required is to set the terminal mode to ON-LINE and the power switch to ON. If your terminal is remote from the system it must use a modem or Data Set to link your terminal to the system. Table 1-1 contains procedures for using most remote terminals.

LINE/LOCAL SWITCH. Nearly all terminals will have a LINE/LOCAL or REMOTE/LOCAL switch. This switch should always be set to the LINE or REMOTE position.

DUPLEX/HALF DUPLEX SWITCH. Some terminals have a FULL DUPLEX/HALF DUPLEX or ECHO/NO ECHO switch or jumper strapping. This should usually be set to the FULL DUPLEX or NO ECHO position. If this setting is improperly made, either the terminal will not print the characters that you type or the characters will be duplicated.

TERMINAL SPEED. Terminals and modems vary in the speed with which they send and receive characters. The system will automatically detect the speed of your terminal and adjust its transmission speed accordingly. The system will accept terminal speeds of 10, 15, 30, 60, 120, and 240 characters per second (14.8 for the IBM 2741). When using a modem with a remote terminal you must use the same speed setting for both the terminal and the modem.

LOGGING ON AND OFF THE SYSTEM

Once connection has been made you must type (*return*) (*line feed*) . This allows the system to determine the speed and parity of your terminal. The system should respond by typing the following message:

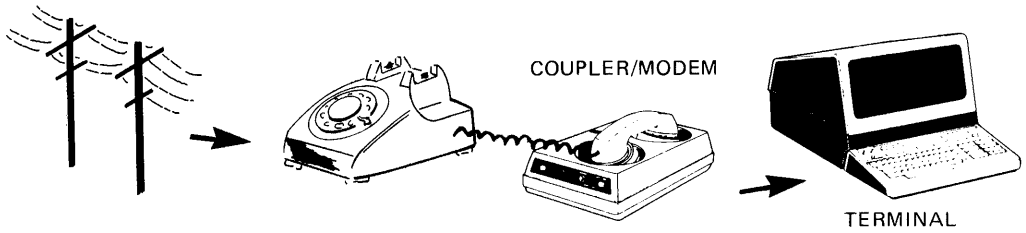
```
PLEASE LOG IN
```

HELLO COMMAND. The HELLO command is used to log onto the system. The command is followed by your account number, password, and terminal type parameter.

```
HELLO-H200,JOHN,1
```

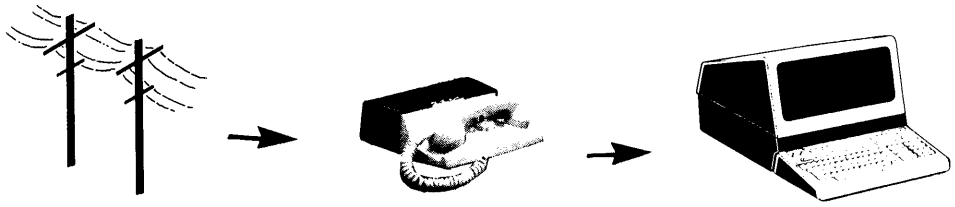
Table 1-1. Remote Terminal Connection Procedures

ACOUSTIC COUPLER MODEM AND TELEPHONE:



1. Set terminal mode to ON-LINE and power swich to ON.
2. Set coupler power switch to ON.
3. If coupler has a duplex switch, set to FULL or FULL UP
4. If coupler has a line switch, set to ON-LINE.
5. Remove telephone handset and dial the computer telephone number.
6. When the computer responds with a high pitched tone, place the handset into the coupler receptacle (the correct handset position should be marked on the coupler).
7. Type (return) (linefeed) and log on.

DATA SET



1. Set terminal mode to ON-LINE and power switch to ON.
2. Press the TALK button on the Data Set.
3. Remove the handset and dial the computer telephone number.
4. When the computer responds with a high pitched tone, press the DATA button on the Data Set; replace the handset in its cradle.
5. Type (return) (linefeed) and log on.

Note: When connection is via telephone lines, the user must log on within a period determined by the system operator (nominally two minutes).

H200 is the account number, JOHN is the account password, and 1 is the terminal type. Note that a comma is used to separate parameters. An account number is assigned to you by the system operator. It consists of a single letter followed by a three-digit number. Your password is linked to your account at the time your account is created by the system operator and can be modified at any time. The password can be from one to six characters. The password may be kept confidential by using non-printing control characters. Control characters are entered by holding the Control key ((*control*) or (*ctl*)) down and then pressing the associated character. Control A is shown as A^c, control Z as Z^c, etc. For example, on a terminal the password $SE^cC^cR^cE^cT$ would appear as ST. Note that certain characters (X^c, H^c, J^c, M^c, and Null) are not allowed as part of a password.

TERMINAL TYPE. The terminal type parameter tells the system what type of terminal you have. This information is used to modify the transmission of data to suit your terminal's requirements. Failure to specify the correct parameter may result in the loss of characters at your terminal or at the system. A list of commonly used terminals and their terminal type parameter is as follows:

0 = HP 2600A, HP 2640A, HP 2749B, or IBM 2741
1 = Execuport 300 or Texas Instruments Silent 700
2 = ASR-37
3 = TermiNet 300, TermiNet 1200, or HP 2762A/B
4 = Memorex 1240

If no value is given for the terminal parameter the system assigns 0 as the default value. If you are unsure of the parameter value for your terminal, consult your system operator.

ERRORS DURING LOGGING ON. If you make a mistake when logging on, the system responds with an appropriate error message. For example, if you forget to type the hyphen while entering the HELLO command:

```
HELLOH200,JOHN,1
```

the system responds with the message:

```
ILLEGAL FORMAT
```

Re-enter the command in the correct form.

If the wrong password is entered:

```
HELLO-H200,JHN,1
```

the system responds:

```
ILLEGAL ACCESS
```

Re-enter the command with the correct password.

The messages ILLEGAL ACCESS and ILLEGAL FORMAT indicate that some or all of the current input is not acceptable to the system.

Spelling mistakes, format errors and incorrect parameters can be corrected while the line is being entered if the error is noticed before return is pressed. The control-H character (H^c) can be used to correct a few characters just typed, or the control-X character (X^c) can be used to cancel the entire line and start over.

Suppose the command HELLO is misspelled during entry. The control-H (H^c) will delete the last character. Depending on the type of terminal you are using, the H^c character may cause an underscore (_) or backarrow (←) to be printed. You can retype the character and finish the line. When you press (*return*) , the line is entered correctly.

```
HELO_LO-H200,JOHN,1
```

If several characters have been typed after the error, the H^c character must be typed for each character to be deleted. In the following example, four characters are deleted:

```
HELO-H2____LO-H200,JOHN,1
```

Another method is to use X^c to cancel the entire line. If you type X^c before you press return, the system responds with a backslash (\ ) at the end of the line and then produces a return and linefeed. The correct command can be entered on the new line:

```
HELO-\
HELLO-H200,JOHN,1
```

All commands including HELLO can be abbreviated to the first three characters.

PROMPT AND SPECIAL CHARACTERS. The system uses prompting and special characters to signal to the user that certain input is expected or that a specific action is completed.

Character	Meaning
?	User input is expected during execution of an INPUT statement.
??	Further input is expected during execution of an INPUT statement.
???	A BASIC command was mistyped; re-enter it correctly.
\	Issued in response to the control character X^c. Indicates that the line being typed just prior to entry of X^c has been deleted.
_ or ←	Issued in response to the control character H^c. Indicates that the character typed just prior to entry of H^c has been deleted.
STOP	Issued in response to the BREAK key. The current operation will be halted.

ECHO COMMAND. The ECHO command can be used to adjust the data transmission to your terminal. If your terminal automatically prints characters that it sends, you can type:

```
ECHO-OFF return
```

to prevent the system from sending out duplicate characters. You can enter

```
ECHO-ON return
```

to cause the system to return data it receives to your terminal. This command is normally only used when your terminal does not have a DUPLEX/HALF DUPLEX switch.

TIME COMMAND. The TIME command can be used to find out how much terminal time has been charged to your account.

```
TIME return
C100 ON PORT #05 FOR 00025 MIN. 00125 MIN USED OF 65000 PERMITTED.
```

MESSAGE COMMAND. The MESSAGE command allows you to enter a line of text at your terminal and have it sent to the system operator. An example of this would be:

```
MES-PLEASE ASSIGN A CARD READER TO C901 return
```

This message will appear on the system console along with a number identifying your port.

BYE COMMAND. When you have completed a session at your terminal, log off the system using the BYE command. For example:

```
BYE return
```

The system will respond by printing the total number of minutes that you were logged on.

```
0014 MINUTES OF TERMINAL TIME
```

You may also log on at the same terminal using the same or another account. This will result in the first account being logged off.

YOU AND THE SYSTEM OPERATOR

There are several procedures available to you as a user which must be performed by the system operator. Table 1-2 contains a partial list of these operations. Depending on the operating policies at your site, you may request the operator to perform certain of these operations. You may use the MESSAGE command to inform the operator of your request or your site may use special request forms. A complete description of each operation available and the related operator procedures is contained in the HP 2000 Access System Operator's Manual (22687-90005).

You can request the system operator to assign additional capabilities to your account. You may then give files in your account the capability of allowing multiple write access. In addition, both your programs and files may be selectively made available to users in other accounts.

Table 1-2. Selected System Operator Commands

COMMAND	THE SYSTEM OPERATOR USES THE COMMAND TO:
ANNOUNCE	Send a one-line message to one or all ports.
ASSIGN	Assign a non-sharable device to one user, to all users, or the RJE facility, or logically remove the device from the system.
BESTOW	Transfer programs and/or files from one user's library to that of another user.
BREAK	Allows a user to use the break key capability of his port after it has been programmatically disabled.
CHANGEID	Modify the capabilities of an existing account.
COPY	Reproduce a copy of a program or file from one user's library in that of another user's library.
KILLID	Remove a specified account from the system.
NEWID	Enter a new account on the system.
PHONES	Reset the time permitted users for logging on.
PURGE	Remove all programs and files from the system library that have not been accessed since a given date.
RESET	Change the total terminal time recorded for a user or all users.

INTRODUCTION TO BASIC PROGRAMMING

SECTION II

This section provides an overview of programming on the 2000 Access System. It discusses programs, the user work space and library, and selected commands. Specialized topics such as arrays, strings, files, formatted output, and system security features are discussed as separate topics in other sections. The material presented in this section serves as the basis for the explanations in the following sections. Complete descriptions of each of the commands and statements discussed in this section are contained in Section X and Section IX respectively.

YOUR WORK SPACE

Each terminal logged on the system is given a work space. This work space is initially empty and will be used to store programs and data during program preparation and execution. Each work space is approximately 10,000 words long.

Since programs are entered one statement at a time, the statements in the work space do not always make up a complete program. The contents of the work space, however, are referred to as the current program whether or not they actually represent an executable program.

You can control your work space in a variety of ways. You can change, display, save, delete, or give a name to whatever is held in the work space. Entering program statements and working with the contents of the work space are discussed in the following paragraphs.

YOUR LIBRARY

Each account has a private library. This library is used to store your programs and data files. The size of the library is set by the system operator. You can make programs and files in your library accessable to other accounts and in addition can access your group and system libraries. (A detailed discussion of libraries and library access is contained in Section VIII.) The remainder of this section will describe your ability to access and modify entries in your own library.

NUMBERS, LOGICAL VALUES, AND EXPRESSIONS

Numbers, logical values and expressions are used in the majority of data processing tasks. The following paragraphs explain how these are defined and used on the 2000 Access system. (String data and text manipulation are described in Section IV).

NUMBERS

Although some real numbers can be represented exactly by the system, others exceed its capacity. The 2000 Access system accepts and uses numbers with magnitudes of -10^{38} to -10^{-38}, 0, and 10^{-38} to 10^{38}. The precision of each number is six or seven decimal digits (23 binary bits). Numbers used or produced by BASIC programs which exceed these limits are converted to the nearest representable number. Numbers that exceed the system's precision are rounded to the nearest six or seven digits. Numbers whose magnitude is less than 10^{-38} are replaced with zero. Overly large numbers are replaced with the maximum magnitude representable on the system. Magnitude violations produce a warning message at your terminal.

LOGICAL VALUES

The system uses two logical values, "1" for true and "0" for false. When tested, any non-zero value (positive or negative) will be interpreted as true. Refer to the discussion of logical operators under Expressions.

EXPRESSIONS

Expressions and in particular numeric expressions are fundamental elements of the BASIC language. Numeric expressions used in BASIC use forms very similar to those used in normal algebraic expressions. This allows you to program most arithmetic operations directly, without changing their form. Even complex calculations can be programmed in a concise manner. Since logical expressions are an extension of arithmetic operations, logical operators can be included in numeric expressions.

An expression is a combination of operators and operands which can be evaluated to a number or logical value. Evaluation consists of applying each of the operators in turn to its associated operands (each of which is associated with a number or a logical value). The number or logical value produced by the last operator becomes the value for the entire expression. For example, the expression 10 − 4 + 2 is evaluated by subtracting 4 from 10 (using the operator "−") to produce 6, and then adding 2 (using the operator "+") to produce the value of the expression 8.

OPERANDS. Operands are the values acted on or tested in an expression. They can be constants, variables, or functions. They are associated with operators according to strict rules (refer to Section XI). In most cases you can easily determine which operators will operate on which operands in an expression. But in cases where you want to ensure that operands will be associated with operators in a specific order you can override the order of association by using parentheses or brackets. For example, while there may be some doubt as to the evaluation of A-B/2, by adding parentheses all doubt is removed, (A-B)/2.

CONSTANTS. A constant has a single numeric value that is retained throughout program execution. Both positive and negative constants can be used in normal decimal notation, and can include a decimal fraction. Very large or very small numbers can be represented in exponential notation. This is a decimal number followed by the letter E and a one or two digit

exponent. The exponent can be signed. For example, .000000005 can be entered as .5E−8, 50E−10, etc. In all cases the value of the constant is the closest approximation that can be represented on the system.

Examples:

```
−25     3.1416    0     98.6E+3
+0.0    1E−38     −0.25     7.7E−07
```

VARIABLES. A variable is a name used to reference a value. Unlike a constant the value associated with a variable can be changed during program execution. The new value can be obtained by evaluating an expression, reading program or file data or requesting direct input from a user. Initially the values of all variables are undefined. The program must assign a value to a variable before requesting it in any operation. An error message will be printed and the program terminated if the program attempts to use an undefined variable. Variables can be of two types, simple or subscripted.

Simple Variables. Simple variables consist of a single letter (A through Z), optionally followed by a single digit (0 through 9). Simple variables are not listed in DIM statements.

Examples:

```
A9    X    C2    Z0
```

Subscripted Variables. It is often convenient to refer to related data together as a set of values. A single letter (A through Z) can be used to refer to a set of values. The same letter can be used to reference both a simple and a subscripted variable. The system identifies the subscripted variable by its following parentheses. Note that you cannot use the same letter for both a one and a two dimensional array. These sets can be arranged as one or two dimensional arrays. Section III contains a detailed discussion of arrays and array operations.

Each individual value in the array can be referred to by a subscripted variable. For one dimensional arrays a subscripted variable is the array name followed by a single integer or expression in parentheses. When an expression is used it is evaluated and rounded to an integer. This integer is used to refer to a specific element in the array. Subscripts can contain subscripted variables.

Examples:

```
A(3), B(X), R(Z**2 +5), T(T(6))
```

In two dimensional arrays individual elements are referred to with two subscripts. The first subscript refers to the row and the second the column position of the element within the array.

Examples:

```
C(I,J), K(3,6), S(Q−1, R+Z), V(R−S(N), T1+T2)
```

FUNCTIONS. There are two types of functions available on the system, predefined and user defined. The names ABS, ATN, BRK, COS, EXP, INT, LOG, RND, SGN, SIN, SQR, TAN, and TIM identify predefined functions. Predefined functions are automatically recognized by the system and are available to any program. Refer to Section XI for a detailed discussion of their use. Up to 26 user defined functions can be defined for use in a program. These functions are referenced by the letters FNA through FNZ. For each user defined function used in a program you must have a DEF statement defining the function.

A function uses one or more values (parameters), performs a specified calculation, and produces a single value as a result. A reference to a function consists of the function name followed by a list of one or more expressions (arguments) enclosed in parentheses. When more than one expression is used they are separated by commas. These expressions are evaluated and passed as parameters to the named expression. The result returned by the function is the value of the function. The number of arguments in a function reference must be the same as the number of parameters defined for the function. A function can be used as an argument of another function.

Examples:

```
FNB(R+T)    ABS(N)    FNC(D0-FND(D0))
SQR(X*X+Y*Y)    FNZ(ABS(FNZ(Z3))/X)
```

OPERATORS. An operator performs a mathematical or logical operation on one or two values, resulting in a single value. Binary operators appear between the two operands whose values they combine. In the expression A+B−C the operands of '+' are A and B but the operands of '−' are A+B and C. Unary operators precede their single operand. For example the expression SQR(−C) negates the value of variable C before taking the square root. There are three types of operators used on the 2000/Access BASIC system: arithmetic, relational, and logical.

Arithmetic Operators. All of the usual arithmetic operations are available in BASIC. The symbols "+" and "−" are used both as unary and binary operators. The context of their appearance determines which operation is intended. Unlike algebraic notation, implied multiplication does not exist in BASIC. Thus A × B must be written as A*B rather than just AB. The operation of raising a number to a power also requires an explicit operator. Thus A^B is written as either A**B or A↑B (the two forms are interchangeable). Two special operators, MIN and MAX, return the minimum or maximum respectively of their two operands. The table below lists the two unary and seven binary arithmetic operators recognized by BASIC.

Operator	Operation	Example
+	Unary add (no effect)	+A
−	Unary subtract (negation)	−A
*	Multiply	A*B
/	Divide	A/B
+	Add	A+B
−	Subtract	A−B
** or ↑	Exponentiate	A**B or A↑B
MIN	Minimum	A MIN B
MAX	Maximum	A MAX B

Relational Operators. Relational operators are normally used to make decisions concerning the flow of control in a program (refer to the IF statement) but are also valid expression operators. They compare the values of two operands to see if they bear the relationship to each other signified by the operator. If the relationship is true, the result value is 'true'; if not, the result value is 'false'. In most programs it is convenient to think of relational operators as producing logical values as their result. However, remember that the value 'true' is actually the number 1 and 'false' is actually the number 0. The table below lists the six binary relational operators.

Operator	Relation Tested	Example
=	Equality	A=B
<> or #	Inequality	A<>B
<	Less than	A<B
>	Greater than	A>B
<=	Less than or equal	A<=B
>=	Greater than or equal	A>=B

Logical Operators. Logical operators are useful in testing multiple relations or performing Boolean operations on 'true' and 'false' values. Like the relational operators, the result of a logical operator is 'true' or 'false' (1 or 0), depending upon the values of its operands. Remember that an operand is considered to be 'false' if its value is 0 or 'true' if its value is not 0. The table below describes the action of the logical operators.

Operator	Result	Example
NOT	'true' if operand is 'false'; 'false' if operand is 'true'	NOT A
AND	'true' if both operands are 'true'; 'false' if either operand is 'false'	A AND B
OR	'true' if either operand is 'true'; 'false' if both operands are 'false'	A OR B

EVALUATING EXPRESSIONS. An expression is evaluated by replacing each variable with it's value, evaluating any function references, and performing the operations indicated by the operators. The order in which operations are performed is determined by the order presented below.

(first)
** ↑
Unary - Unary + NOT
* /
+ -
MIN MAX
< <= = >= > <> #
AND
OR
(last)

The first operator applied within an expression is the one highest in the above table. Other operators follow in turn according to their relative priority. When two or more operators of the same level appear, they are applied in left-to-right order. If this order of evaluation is not the one desired, parentheses can be used to override it. Any part of the expression enclosed in parentheses (which must itself be a legal expression) is evaluated completely as an operand of the surrounding expression. Parentheses can be nested to any desired degree to force an intended order of evaluation. The following pairs of expressions are identical in their evaluation.

```
A+B*C**D MAX E/D-F                  (A+(B*(C**D))) MAX ((E/D)-F)
A < -B*C OR NOT D AND E>F           (A<((-B)*C)) OR ((NOT D) AND (E>F))
A/B/C AND D*E*F AND G**H**I         (((A/B)/C) AND ((D*E)*F)) AND ((G**H)**I)
```

The general form of an expression is an alternation of operands and binary operators, where each operand can be preceded by a single unary operator. The only restriction to this rule is that the binary operator ** (or ↑) cannot have a unary operator following it. The same effect could be accomplished by enclosing the desired unary operator and its operand within parentheses. For example, A**(-B).

Examples:

```
-A0/3.2      B**N+B**(N+1)      3.1416*R**2      (C MAX 100)+D
A =0      (C0*C1)      (D0/D1)      SQR(ABS(R5))
T1 AND T2 OR (NOT T1 OR NOT T2)      FNM(K(I)-K(J))/ABS(I-J)
```

PROGRAMMING

The 2000 Access system is designed for interactive use at a keyboard terminal. Statements, commands, and data for executing programs are entered without strict format rules. Except where noted the system ignores blanks. This means that you can use blanks to make statements or input easy to read or leave them out entirely. Each command and statement is tested for errors in syntax, obvious logical mistakes, and violations of system security. Errors or violations are rejected and a message is returned to you to aid in isolating and correcting the problem.

BASIC programs are made up of one or more statements. The statements are used to perform or define operations on data or to add narrative comments to the program. Each statement must be preceded by a statement number (an integer between 1 and 9999). The statement number is used to determine the program's order of execution and to provide a means for one statement to reference another. Program execution begins with the lowest numbered statement. Since statements are not executed as they are entered you do not have to enter statements in the order in which they are to be executed. When a program is complete you can use the RUN command to execute it.

Each statement is entered as a separate line ended with (*return*). The system will either accept the line, responding with a (*line feed*) or reject the line with the message ERROR. A short description of an error can be obtained by entering any character other than (*return*) following the ERROR message. Since rejected statements are not entered into your work space

correction consists of retyping a correct statement on a new line. If you detect an error in a statement before you have typed (*return*) you can use the control-H (H^c) or control-X (X^c) characters to alter or cancel the line as described in Section I. Statements already entered can be replaced by entering a new statement with the same statement number. Entering only a statement number followed by (*return*) deletes the numbered statement from the work space.

ASSIGNMENT Statement

The assignment statement (also called the LET statement) assigns the value of an expression to one or more variables. As the examples show, LET is an optional part of the assignment statement which can be used for consistency or clarity. When a subscripted variable is to be assigned a value, its subscript is evaluated first and then the expression. Thus A(I) = I = (T+R)/S is equivalent to I = A(I) = (T+R)/S. There is potential confusion between the use of "=" as an assignment operator and its use as a relational operator. The rule is that, proceeding from left to right in the statement, an "=" is an assignment operator unless it violates the format of the assignment operator in which case it and any following are relational operators. Consult Section XI if a more rigorous explanation is needed.

Examples:

```
LET A = I+5
A = I+5
A(1) = A(2) = A(3) = 0
LET C1 = C2 = A+B=T-R
Q = FNC(C0) MAX FNC(C1)
```

GO TO Statement

The GO TO statement alters the normal control flow of a program by specifying which statement is to be executed next. The statement number selected must be a valid statement in the program. While statements of the form 10 GO TO 10 are legal, once executed they cause the program to loop endlessly. The GO TO statement can specify a single unconditional destination or it can contain an expression selecting one of a list of statement numbers. The statement numbers are numbered sequentially, beginning with 1, in left to right order. The value of the expression is rounded to an integer and control is transferred to the statement number in the corresponding position. If the integer is less than 1 or greater than the number of statement numbers in the list, then the GO TO is ignored and control passes to the next statement in the program.

Examples:

```
GO TO 100
GO TO N OF 1000,2000,3000
GO TO (P0-Q0)/5 OF 10,20,30,40,50
```

END and STOP Statements

The last statement of a program must be an END statement. When an END statement is reached, the program's execution is terminated and control returns to the user. If other places within the program are logical points of completion, they can either transfer control to the END statement with a GO TO statement or execute a STOP statement, which is identical in effect to an END statement. The STOP statement is preferred in these cases because it provides a visual reminder of the program logic.

Examples:

```
 200 STOP
1000 END
9999 END
```

IF Statement

The IF statement provides a means of testing for a specific condition and transferring control to another statement if the condition is 'true'. The expression following the word IF is evaluated and if 'true' (or non-zero) then control passes to the statement number following the word THEN. This statement number must be a valid statement existing in the program. The IF statement is a powerful tool for choosing alternative logic paths, controlling program loops by specifying a completion condition, or making any simple or complex decision needed to accomplish a program's purpose.

Examples:

```
10 LET I = 1
20 A(I) = I*I
30 LET I = I+1
40 IF I<=10 THEN 20
50 END
```

This program computes the squares of the numbers 1 to 10, stores them into the array A, and terminates.

```
20 IF A<B AND C<D THEN 200

30 IF NOT T1 THEN 45
```

PRINT Statement

The PRINT statement provides a simple and direct means for BASIC programs to display values on your terminal. The items to be printed (print list) are evaluated and displayed in sequential order from left to right in one or more lines. Each item in the print list is separated from the following one by either a semicolon or a comma. A semicolon serves only to separate items in the print list; the first character of the following item's value is printed immediately after the last character of the preceding item's value. A new line is begun whenever the next value will not fit on the remainder of the current line.

A comma serves another purpose besides separating print list items. Each line on the terminal is divided into four consecutive fields of 15 character positions and a final field of 12 character positions (72 character positions per line). When a comma follows an item which did not completely fill the last of the one or more fields its value occupies, blanks are added to complete that field. Thus the value of an item following a comma always begins at the start of a field. If the previously completed field was the last one in the current line, the next item's value will be printed on a new line. Since most values occupy less than fifteen character positions, the usual effect is a table of five columns per line.

If a comma or semicolon follows the last item in the print list, the print list of the next PRINT statement executed will be printed as a continuation of the current line. Otherwise, the current line is completed and the next PRINT statement will begin printing values on the next line. A PRINT statement with no item list either completes the previous line, if it ended with a comma or semicolon, or skips the current line. This is frequently used to produce a blank line.

PRINTING EXPRESSIONS. If a print list item is an expression, its value is displayed as a number. The first character of a number is either a minus sign (−) if the number is negative or a space (implied plus sign) if it is positive. Integral values less than 32768 in magnitude are displayed as integers. For other values the first six decimal digits are printed with a decimal point in the appropriate position. If the magnitude of the number is less than .09999995 (and cannot be displayed exactly with six digits including the zeros following the decimal point) or exceeds 999999.5, six digits of precision are printed as a number greater than 1 but less than 10, followed by an exponent. This notation is interpreted as explained under CONSTANTS. The exponent always appears as a two-digit signed integer following the letter E. For example, 12345600 is printed as 1.23456E+07 with a preceding blank as the implicit plus sign of the number.

Each number, when printed, is followed by at least one blank. Additional blanks are appended if necessary to make the total number of characters (beginning with the sign) either 6, 9, 12, or 15. Thus even when expressions are separated by semicolons, at least one blank will always appear between the end of one number and the sign character of the next. Leading zeros are never printed and trailing zeros following a decimal point are printed as blanks unless the number form includes an exponent (in this case all six digits of precision appear). Refer to Section XI for a detailed explanation of how numbers are printed.

Examples:

```
10 PRINT 4.0 ; -.02500000 ; -.02500001 ; 999999.4

 4     -.025         -2.50000E-02     999999.


10 PRINT  1;-10;100;-1000;10000;-100000
20 PRINT  1,-10,100,-1000,10000,-100000

 1     -10      100   -1000       10000    -100000.
 1                -10                 100              -1000              10000
-100000.
```

PRINTING LITERAL STRINGS. It is often desirable to print textual information along with numbers. This facility can be used to identify values, produce table headings, and otherwise clarify the results of a program. BASIC provides another type of print list item, called a literal string, for this purpose. A literal string is simply a sequence of characters enclosed by double quote marks ("). Blank characters in quotes are not ignored, but are part of the literal string. The minimum number of characters in a literal string can be as few as zero. The practical maximum is 72 (refer to Section XI for exceptions to this rule). Note that a double quote mark itself cannot appear in a literal string. Section IV describes a more general form of literal string which can contain double quote marks and other special characters.

A literal string is displayed without the delimiting quote marks. Since no leading or trailing blanks are added, the number of character positions occupied by a literal string is simply the number of characters in the string. As a result, if a semicolon separates consecutive literal strings in a print list, the second string will immediately follow the first without any additional intervening blanks. If a comma follows a literal string then the last of the fields it occupies will be filled with blanks. Although two expressions in a print list must always be separated by a comma or semicolon, a literal string need not be separated from either the preceding or following print item. If the punctuation is omitted, BASIC assumes an implied semicolon. If during execution of a PRINT statement a literal string will not fit into the current line, then it begins in the next line.

Example:

```
10 LET A = -1
20 LET B0 = 0
30 LET B1 = 1
40 PRINT "A="A"B0 AND B1 ARE"B0;B1
50 END


A =-1     B0 AND B1 ARE 0       1
```

PRINT FUNCTIONS. Up to this point the comma separator provides the only control over the format of data displayed by a PRINT statement. Although a method of requesting completely precise formatting is described in a later section, BASIC also contains some additional simple control facilities. Three special predefined functions, TAB, SPA, and LIN, are accepted as special print list items. They are referred to as print functions since they can only appear in a PRINT statement. As for expressions, they must be separated from any other print list item (except a literal string) by either a semicolon or comma. Unlike expressions, however, a following comma does not have any spacing effect; it is equivalent to a following semicolon for the print functions. All three expect a single expression as their argument. When encountered in a print list, the expression is evaluated and rounded to an integer value.

The TAB function provides horizontal tabulation. The 72 character positions in a line are numbered from 0 to 71. This function generates enough spaces (blank characters) to fill the current line up to the character position specified by its argument. If the current line is already at or past that position, no action is taken. If the argument exceeds 71 then the current line is simply completed and the next print item will begin on the next line. The SPA function generates the number of spaces indicated by the value of its argument. If the number of spaces specified is greater than 71 or will not fit on the current line, then the current line is simply completed. If the value of its argument is positive, the LIN function completes the indicated number of lines. For example, LIN(3) completes the current line (which could be empty) and then skips two more lines. If the value of its argument is negative, LIN advances the indicated number of lines but preserves the current character position. For example, LIN(−1) tabulates one line vertically (printing continues at the current horizontal position but on the next line).

Example:

```
10 PRINT "ABC";LIN(-1);"DEF";LIN(2);"GHI";SPA(3);"JKL"
20 PRINT TAB(5);"MNO"
30 END


ABC
   DEF

GHI   JKL
     MNO
```

READ/DATA/RESTORE Statements

It is sometimes desireable to set a large number of variables to different constant values. This cannot be done with assignment statements unless you write one statement for each constant. However, the READ, DATA, and RESTORE statements provide a simple means to accomplish this and other similar programming tasks. DATA statements contain a list of constants. All the constants in the collection of DATA statements comprise a data list. The list starts with the DATA statement having the lowest statement number and continues with successively higher numbered DATA statements. DATA statements can appear anywhere in the program and do not need to be consecutive. DATA statements are not executable. If control is passed to one of them, the next executable statement is executed with no other effect.

A pointer is kept within the program to indicate which constant is next in the list. At the beginning of a program's execution, the pointer is set to the first constant in the first DATA statement. When a READ statement is executed, each variable to be read is assigned to a new value from the constant list. Both simple and subscripted variables can appear in the list of a READ statement. The pointer advances consecutively through the constant list as each assignment is made. The RESTORE statement resets the pointer to the first value, allowing the constant list to be reused. If the RESTORE statement includes a statement number, the pointer is set to the first constant in the referenced DATA statement. If the specified statement is not a DATA statement, the pointer is set to the constant list of the first DATA statement whose statement number is greater than the one given. An error occurs if a READ statement requests more values than remain in the list. The program terminates with an appropriate message.

Note that the ease of modifying a program can be utilized to advantage. Once a general program has been written to perform a desired task, it can be supplied with different data for each execution by entering new DATA statements with the same statement numbers used before.

Example:

```
 10 I = S = 0.0
 20 READ T
 30 IF T=999999 THEN 70
 40 I = I+1
 50 S = S+T
 60 GO TO 20
 70 PRINT "AVERAGE IS"S/I
 80 IF I>=3 THEN 100
 90 STOP
100 RESTORE
110 READ A,B,C
120 PRINT "AVERAGE OF FIRST THREE IS";(A+B+C)/3
500 DATA 5.5, 3.46, 52
510 DATA 77.3, .89
600 DATA 999999
999 END
```

Assuming that a list of numbers is supplied in DATA statements having statement numbers greater than 120 and less than 600, this program will compute their average without knowing how many there are. If at least three exist, it will also compute the average of the first three.

FOR and NEXT Statements

Frequently, programs need to perform the same set of operations several times, where the data used in each iteration has a computable relationship to the previous iteration. Although a combination of the IF and GO TO statements usually solve this problem, the resulting program is often difficult to understand. The two statements FOR and NEXT allow repetition of a group of statements in a clearly understandable fashion. The FOR statement precedes a group of statements to be repeated and specifies the number of iterations desired. The NEXT statement directly follows the group of statements, acting as an implicit GO TO to the first statement in the group after each iteration. You can use FOR/NEXT pairs within other FOR/NEXT pairs to produce loops within loops. No loop, however, can be partially contained within another. Also, the loop control variables of nested loops should not be the same as the control variables of outer loops.

When a FOR statement is executed, the variable preceding the assignment operator, called the control variable (it cannot be a subscripted variable), is assigned the result of evaluating the expression which follows. At this time the limit value (given by the expression following TO) and step value (given by the expression following STEP, or assumed 1 if not indicated) are also evaluated. The statements following the FOR statement, and preceding the closest NEXT statement referencing the same control variable, are then executed none or more times according to the steps below.

1. If the value of the control variable exceeds the limit value (or is less when the step value is negative), control passes to the statement following the associated NEXT statement. Otherwise proceed to step 2.
2. Control passes to the statement following the FOR statement.
3. When control reaches the NEXT statement, the step value is added to the control variable's value.
4. Repeat from step 1.

A FOR/NEXT loop terminates normally when the value of the control variable exceeds that of the limit value. Programs often terminate loops by transferring control out of them with an IF statement, GO TO statement, etc. This causes no difficulty and the program can later reuse the loop by transferring control to the FOR statement. However, it is very bad practice to transfer control into a loop (to any statement following the FOR statement, up to and including the NEXT statement). When control reaches the NEXT statement unpredictable results can occur.

Example:

```
100 FOR I = 1 TO 5
200 FOR J = I TO 1 STEP -1
300 PRINT J;
400 NEXT J
500 PRINT
600 NEXT I
700 END

1
2   1
3   2   1
4   3   2   1
5   4   3   2   1
```

INPUT Statement

The INPUT statement allows you to input data to an executing program from your terminal. An INPUT statement contains a list of variables (simple or subscripted) which are to receive new values. When the program executes an INPUT statement it prints a question mark (?) at the terminal to signify readiness for the data and then waits for a response. The response is one or more constants (in the same form described under CONSTANTS) separated by commas. The constants are assigned to the variables in the input list in left to right order. Subscripts are not evaluated until the variable containing them is to receive a value (thus INPUT A(I),I will not in general produce the same results as INPUT I,A(I) even if you reverse the order of constants in the response). If an insufficient number of constants is typed, the program responds with a double question mark (??) to tell you that more data is required for that INPUT statement. If a constant is not legal, a message informs you that you should reenter part of the list, beginning with the correct version of the erroneous constant. If too many constants are contained in the response, you are notified and the extra ones discarded. Rather than respond to a program waiting for input, you can terminate its execution by pressing the *break* key.

Example:

```
10  PRINT "WHAT ARE YOUR NUMBERS";
20  INPUT A, B
30  PRINT "THEIR SUM IS";A+B
40  END
RUN


WHAT ARE YOUR NUMBERS? 5, 7
THEIR SUM IS 12

DONE
```

Note that the question mark is printed at the current line position.

ENTER Statement

The ENTER statement provides the program with more control over the input operation. The statement can limit the amount of time allowed to respond with data, provide the program with the actual time taken to respond, indicate whether the data was acceptable, and return the port number of the user's terminal. The port number can be obtained separately, without involving the user at all, or together with a single data value. Data can also be requested without asking for the port number. If a number sign (#) follows the keyword ENTER, then the variable it precedes is assigned the user port number (an integer in the range 0 to 31). Otherwise (or following the port return variable) the first expression is evaluated and rounded to an integer (which must be in the range 1 to 255) specifying the number of seconds permitted the user to respond. No prompt is printed, the program must notify the user that input is expected by a message in a preceding PRINT statement. The variable following the expression is set to the approximate time, in seconds, that the user took to respond. If the constant was not legal, the time is negated. If the allotted time has elapsed, the value −256 is returned; the values −257 and −258 indicate a transmission problem occurred and the user should be asked to respond again. The last variable in the list is assigned the single value which the user is expected to enter. Unlike the INPUT statement, ENTER does not respond to the (*return*) (which completes the user's answer) with a (*line feed*).

Examples:

```
10 ENTER #P
20 ENTER 255,R,A
30 ENTER #P,T1,R,A(I)
```

REM Statement

BASIC is designed to simplify the translation of computational tasks into computer programs. It is not always equally simple to deduce the task performed by examination of a program's text. External documentation can help, but it is still important to insert explanatory comments at key points of the program itself. The REM statement allows the user to include remark lines as part of the program. The form of the statement is merely REM followed by any text. All characters appearing after the REM are considered to be a part of the remark. Blank characters are not ignored and special characters such as quote marks are permitted. A REM statement can appear anywhere in the program and does not affect its execution in any way. If control is passed to a REM statement, it continues to the following statement with no other effect.

Examples:

```
10 REM THIS IS AN EXAMPLE
20 REM ANY CHARACTERS, SUCH AS " OR +, ARE ALLOWED.
30 REMARK 'ARK' ARE THE FIRST THREE CHARACTERS OF THIS REMARK
```

GOSUB and RETURN Statements

Some programs use subroutines (groups of statements) to perform an action needed at several locations. It would be inefficient to insert a copy of these statements at each point where their effect is desired. The GOSUB and RETURN statements together allow you to transfer control to another portion of the program and remember where the transfer originated. The GOSUB statement either specifies a single unconditional destination or a list of statement numbers and an expression whose value is used to select one of them. (Refer to the GO TO statement for an explanation of the selection process.) In addition the statement number of the statement following the GOSUB statement is saved. When the sequence of control reaches a RETURN statement, control returns to the statement whose statement number was saved. Thus a subroutine can be written as a group of statements followed by a RETURN statement and it can be called or referenced with the GOSUB statement.

Subroutines can be nested during execution; that is, up to twenty GOSUB statements can be executed without intervening RETURN statements. However, there is no fixed association of RETURN statements with GOSUB statements. When a RETURN statement is executed, control returns to the statement following the most recently executed GOSUB statement. There is no way to return to the source of a transfer until returns have been made from any subsequent GOSUB statements. Of course, subroutines are nothing more than collections of statements and direct transfers using the GO TO or IF statement can be made into or out of them at any point. However, the user should arrange groups of statements intended to be used as subroutines, and limit the flow of control to clearly display their purpose. Careless use of this facility will make programs that are difficult to understand.

Example:

The following program portions use a subroutine to ask the user for the time of day.

```
 200 L = 23
 210 PRINT "HOUR OF DAY";
 220 GOSUB 1000
 230 H = R
 240 L = 59
 250 PRINT "MINUTE OF HOUR";
 260 GOSUB 1000
 270 M = R
 280 PRINT "SECOND OF MINUTE";
 290 GOSUB 1000
 300 S = R
   :     :
1000 REM BEGIN SUBROUTINE
1010 INPUT R
1020 IF R<O OR R>L THEN 1040
1030 RETURN
1040 PRINT "IMPOSSIBLE, CORRECT VALUE IS";
1050 GO TO 1010
1060 REM END SUBROUTINE
```

DEF Statement

The DEF statement is the means by which a program creates a user defined function. Up to 26 functions (FNA through FNZ) can be used. A program which includes references to a user defined function must contain a DEF statement defining it. Failure to do so will result in an error. DEF statements can appear anywhere in the program. If control is passed to one of them, it is passed to the following statement with no other effect. Execution occurs only by evaluation of a reference to the function. Defining the same function twice will result in an error.

The result of a function reference is obtained by evaluating the expression following the equals sign (=) in the appropriate DEF statement. Each user defined function has exactly one parameter, designated by the simple variable name which appears in parentheses following the function name in the DEF statement. The value of an argument (the expression appearing in a function reference) is assigned to the parameter before evaluating the defining expression. The parameter name can also appear as a simple variable elsewhere in the program.

Any legal expression can be used to define a function. Expressions can include references to other user defined functions. However, you should exercise caution to avoid circular definitions. The simplest example is a DEF statement whose expression contains a reference to the function being defined. A more subtle case is where two functions are defined in terms of each other.

Examples:

```
DEF FNT(X) = SIN(X)/COS(X)    defines FNT as the tangent function
DEF FNC(R) = 3.1416*R**2      computes the area of a circle from its radius
```

COMMANDS

Commands instruct the system to perform one of a large number of possible control or utility tasks. Commands differ from BASIC language statements in form and result. A statement is always preceded by a statement number and, when entered, becomes a part of the current program in your work space. A command is not preceded by a statement number and, when entered, instructs the system to perform its indicated action immediately. Commands can be entered at any time except when the current program is executing. They are either accepted and executed or rejected with an appropriate message.

Each command is a single word which can be abbreviated to its first three letters. Some commands also have optional or required parameters following them. If parameters are used, they are separated from the command name by a hyphen (-). Multiple parameters are separated from each other by commas. The user signals completion of a command's entry by pressing (*return*). If the command is misspelled or otherwise unrecognized, the system responds with three question marks (???).

Many commands do not produce a response, their completion is announced with a simple (*line feed*). Others display one or more lines of information. In these cases the user can abort the command's operation by use of the *break* key. The message STOP indicates that the terminal is again available for entering statements or commands.

PROGRAMMING UTILITY COMMANDS

Several commands are especially useful during program development. They perform various utility operations on your work space, usually to create required conditions for other commands.

NAME Command

Your work space can be given a name, consisting of one to six letters and digits. If present, the name is printed as a heading for the output of some commands (for example, LIST and RUN). The work space must have a name in order to save its contents in the account library. If the work space already has a name, this command replaces it with the one specified. The command can also be used without a parameter to erase the current work space name. In this case the hyphen is optional.

Examples:

```
NAME-PROG1
NAM
NAM-123456
```

RENUMBER Command

This command is used to renumber all or a portion of the statements in the current program. Statements containing references to other statements (GOTO, RESTORE, etc.) are automatically changed to reflect the new statement numbers unless they refer to non-existent statements. This command does not alter the logical relationships of the current program or change the order of statements. Renumbering is particularly useful after substantial editing or to establish statement number conventions for portions of the current program. It is also frequently necessary to renumber a program in preparation for use of the APPEND command.

The RENUMBER command has four optional parameters. They are interpreted in the following order (default values are enclosed in parentheses): initial new statement number (10), interval between new statement numbers (10), first statement number of the portion to be renumbered (first statement number of the current program), and last statement number of the portion to be renumbered (last statement number of the program). If any parameter is omitted, then all following parameters must also be omitted. The values of the third and fourth parameters specify a range, all existing statement numbers in the range are renumbered. It is not required that these values match existing statement numbers in the current program. The command is rejected if the first value of the range exceeds the second one, if the requested renumbering would require some statement to be given a statement number greater than 9999, or if renumbering the portion specified would cause the range of its statement numbers to overlap with those of statements outside the portion specified.

Examples:

```
RENUMBER-100
REN
REN-500,10,301,399
REN-1,1
```

DELETE Command

Groups of one or more statements can be erased from the current program with the DELETE command. The two parameters specify a range; all statements with statement numbers in that range are deleted. If the second parameter is not given, it is assumed to be 9999 (all statements with statement numbers greater than or equal to the first parameter are deleted). If both parameters have the same value, only the indicated statement is deleted. Note that this can be done more easily by entering the statement number followed by a (*return*).

Examples:

```
DELETE-100,199
DEL-8000
```

SCRATCH Command

The SCRATCH command deletes the entire current program and erases the current name of the work space. This restores the work space to the state which existed just after logging on. This command is normally used in preparation for beginning development of a new program.

Examples:

```
SCRATCH
SCR
```

MODIFYING THE ACCOUNT LIBRARY

You can alter the contents of your account library at any time. The current program can be saved (if the library space allotment permits) in either of two forms. Existing library entries can be deleted when no longer needed. You cannot use these commands to alter the contents of your group library or the system library.

SAVE Command

A copy of the current program is saved under the current name of the work space. The contents of the work space are not altered. The current program does not have to be complete (executable). Thus the SAVE command is useful for preserving a partially developed program prior to logging off the system. It is also useful for saving intermediate versions of a program. If an editing mistake or other problem ruins the current program, an earlier version can be retrieved. The command is rejected if the work space does not have a name, if the name duplicates that of an entry already in the account library, or if there is insufficient space in the library to accommodate the current program.

Examples:

```
SAVE
SAV
```

CSAVE Command

Like the SAVE command, the CSAVE command saves a copy of the current program in the account library. Prior to making the copy, the program is partially 'compiled'. That is, the first stages of execution are performed. A program stored in this form will occupy slightly more library space than if stored with the SAVE command. However, it will also execute faster when retrieved since the preliminary processing has already been done (this is useful when executed as a result of the CHAIN statement, discussed in Section VII). Unlike the SAVE command, the current program must be complete and executable in order to use the CSAVE command. The command will be rejected if the pre-execution processing detects an error in the program or if any of the conditions mentioned under SAVE exist.

Examples:

```
CSAVE
CSA
```

PURGE Command

The PURGE command deletes the named program from the account library. It does not affect the current work space. A purged program cannot be recovered (unless another copy exists). The amount of space it occupied is returned to the account library. The PURGE command is frequently used to delete an obsolete version of a program so that a new one can be saved under the same name.

Examples:

```
PURGE-PROG1
PUR-123456
```

LOADING THE WORK SPACE

Several commands are available for placing entire programs or major portions into the work space at one time. These rely on the existence of a previously prepared, external version of the statements to be entered.

GET Command

The GET command retrieves the named program from an account library. The command first performs an implicit SCRATCH, clearing the work space and its name, and then loads a copy of the designated program into the work space, setting the name to correspond. The account library is not altered. The GET command can also obtain programs from account libraries other than your own. To retrieve a program from your group library, prefix the name with an asterisk (*). To retrieve a program from the system library, prefix the name with a dollar sign ($). Security provisions may prevent retrieving some programs from these libraries or impose conditions on the copies obtained, such as prohibiting you from saving your own version. (Refer to Section VIII for details.) You can always retrieve any program in your own account library. Once it is in the work space, it is indistinguishable from a program entered by any other means.

Examples:

```
GET-MYPROG
GET-*GROUP1
GET-$LIBPRG
```

APPEND Command

The APPEND command appends a copy of a program from an account library to the current program. Note that the current program is extended, not altered, and the name of the work space remains unaffected. Library entries are not restricted to being complete programs. The APPEND command is frequently useful for incorporating previously prepared utility subroutines into a program under development. The first statement number of the retrieved program must be greater than the last statement number of the current program. The command is rejected if this is not true. The RENUMBER command may be used in most cases to achieve this relationship.

Not all programs can be appended. Those stored with the CSAVE command are not acceptable. A group or system library program can be appended (by prefixing its name with * or $ respectively) unless security provisions prevent it. Security considerations can also impose conditions of use on the composite program if the source account library is not your own.

Examples:

```
APPEND-LOCAL
APP-*GSUB
APP-$SYSUTL
```

TAPE Command

Groups of statements or complete programs on paper tape can be entered into the work space from terminals equipped with a paper tape reader. Some terminals provide the equivalent capability with magnetic tape cassettes. Terminals behave differently when transmitting data from their tape reader rather than the keyboard. The TAPE command causes the system to withhold printing of any diagnostic messages and suppresses the (*line feed*) response to each line. Note that statements entered under control of a TAPE command are merged with the current program just as if they came from the keyboard. Thus the work space should be cleared with the SCRATCH command if a complete program is to be entered.

A particular sequence of actions must be followed to use this facility. Mount the tape in the reader and turn it on before typing TAPE and (*return*). This order of operation is important since the system responds to receipt of a TAPE command with a special character intended to initiate reading. The system assumes that receipt of a command implies that the tape has been completely read. If any errors occurred during entry of the statements, they are printed at the terminal and the command is ignored.

Examples:

```
TAPE
TAP
```

KEY Command

The KEY command is used to terminate the effects of a TAPE command. Upon its receipt, the system returns to keyboard conventions for terminal interaction. This command also causes printing of any diagnostic messages withheld while reading the tape. Any other command received while under the influence of a TAPE command has the same effect. However, commands substituted for KEY in this way will not be executed if any diagnostic messages are waiting to be printed.

Example:

```
KEY
```

EXECUTING AND REPRODUCING PROGRAMS

As mentioned previously, programs are not executed when entered. Instead, the system provides a command to initiate execution of the program in the work space. As a convenience, another command combines the requests for loading and executing a program saved in an account library. Additional commands allow display or reproduction of the current program.

RUN Command

The RUN command initiates execution of the current program. Execution normally starts with the first statement of the program. However, an optional parameter can be supplied to override this convention. If used, execution begins at the indicated statement number, or the next greater one if the specified statement number does not exist. An executing program can be terminated with the *break* key unless this capability has been disabled. The command is rejected if the program is incomplete (for example, lacks an END statement). Execution is usually terminated if an error occurs; however, some questionable conditions merely produce a warning. A diagnostic message indicates the cause of rejection or abnormal termination. After execution ceases, independent of the reason, the program remains in the work space.

Examples:

```
RUN
RUN-100
```

EXECUTE Command

The EXECUTE command acts as a combination of the GET, RUN, and SCRATCH commands. It clears the work space, retrieves the specified program, and initiates execution. Execution always begins at the first statement. Execution of programs from a group or system library is requested by prefixing the name with an * or $ respectively. After execution ceases, irrespective of the cause or the source of the program, the work space is again cleared. The EXECUTE command is provided for reasons other than simple convenience. The system security provisions define programs which can be executed by this command but not loaded by the GET command. Programs in your account library are never restricted in this manner.

Examples:

```
EXECUTE-MYPROG
EXE-*GROUP1
EXE-$SYSPRG
```

LIST Command

The LIST command displays the current program at your terminal. Statements are listed one per line in ascending order of statement number. The system does not reproduce the textual form of a program as it was entered. Instead, statements are displayed with the minimum number of blanks needed for clarity. Within a statement numeric constants appear in one of several standard forms, similar to those produced by the PRINT statement. To emphasize the presence of subscripted variables, the system displays their parentheses in the alternative form of brackets ([and]). The LIST command is most useful during program development. The ease of editing and adding new statements permits substantial alteration of the current program over even short spans of time. You can check the cumulative effect of a series of changes by requesting a listing.

Normally the LIST command reproduces the entire current program. However, optional parameters (separated by a comma) can be used to specify the beginning and end of a range of statement numbers. In this case only statements within the indicated range are listed. If the first parameter appears alone, listing begins with the indicated statement number and continues to the end of the program. If the second parameter appears alone, it must be preceded by a comma to distinguish it from the previous case. Listing begins with the first statement of the program and continues up to and including the indicated statement number. A third optional parameter, the letter P, produces six blank lines after every fifty six lines of statements.

If your terminal prints on paper, the resulting listing can be cut into individual pages. A P can follow the range parameters (preceded by a comma) or, in their absence, follow the hyphen. This option is useful for preparing archival copies of a completed program.

Examples:

```
LIST
LIS-P
LIS-,500,P
LIS-100,299
LIS-8000
```

PUNCH Command

The PUNCH command is a special form of the LIST command. It is useful only for terminals with an auxiliary paper tape punch. All of the parameters available for the LIST command are interpreted in the same manner by the PUNCH command. The current program can be reproduced on the terminal's punch unit by turning on the unit (ensure that the tape is properly mounted) and issuing the command. The program is listed at the terminal at the same time as it is punched onto the tape. Note that the program name followed by leading null characters (called leader) are punched preceding the program and additional null characters (called trailer) are punched after it. Each statement is terminated with the characters X-OFF, (*return*), and (*line feed*). This convention permits the tape to be read by an executing program (in response to INPUT or ENTER statements) or by the TAPE command. Although it is more convenient to save programs in your account library, copies made by the PUNCH command are useful for programs whose infrequent need does not justify their retention in the library space.

Examples:

```
PUNCH
PUN-50
PUN-500,700,P
```

STATUS COMMANDS

Several commands display different kinds of status information. One of these, the TIME command, is discussed in Section I. The others describe the library entries available to the account and report several resource statistics.

LENGTH Command

The LENGTH command displays the length of the current program, the size of your account library, and the library space authorized for your account. The length of the current program is given both in words and records. A record is the basic unit of measurement for library space. The program length in records indicates how much of the remaining library space would be used if the current program were saved. The user work space has over 10,000 words. The program length in words indicates how much of this space is currently occupied. An executing program requires additional portions of the work space for the values of variables and various internal tables. The amount needed is difficult to predict and most users will not find it worth while to attempt to evaluate it.

Examples:

```
LENGTH
LEN
```

Example of system response:

```
01253 WORDS = 05 RECORDS. 00864 RECORDS USED OF 10000 PERMITTED.
```

CATALOG, GROUP, and LIBRARY Commands

These commands are used to print an alphabetic list of the entries in an account library. CATALOG lists the contents of your account library, GROUP lists the group master's library, and LIBRARY lists the system library. An optional parameter can be used to indicate a starting point. The list will begin with the entry named or, if no corresponding name exists, with the first name alphabetically greater. For example, CAT-S will display all entries whose names begin with S through Z. The (*break*) key can be used to terminate the listing. When the GROUP or LIBRARY command is used, only entries accessible to you will appear (refer to the discussion on security for details).

The list contains the name, appropriate code letters, the length in records, and a record length in words for each entry. The last of these is only applicable to data files and will not be discussed here. The length in records is the amount of library space occupied by the entry. It is the same value which would be given by the LENGTH command if the program were loaded into the work space with a GET command. Two code letter positions appear between the name and the length in records. The first of these is a blank for programs stored with the SAVE command or a C for programs stored with the CSAVE command. The letters A, F, and M are used to indicate entries which are data files (discussed in later sections). The second code letter indicates any security restrictions on the entry. It will normally be blank for entries in your account library.

In listings of the group or system library, the letter U indicates that you have unrestricted access to the entry. The program can be loaded with the GET command, appended with the APPEND command (unless it was stored with CSAVE), or executed with the EXECUTE command. Refer to Section X for an interpretation of the letters L and P.

Examples:

```
CATALOG
CAT-PROG1
LIBRARY
LIB
GROUP-A
GRO-222
```

Examples of system response:

```
LIB
 NAME     LENGTH RECORD   NAME     LENGTH RECORD   NAME     LENGTH RECORD
AAA    FP      2          AB     MP   230          BAA    A    LP2     66
BAB     P     13          BAC            6          BAD    C     18
BB     FU     46          BBB    FL     46   128    BFILE FU     128
BUDGE         12          BUDGEU M      12          CR     A     CR0     40
CT     A     230          D      F     100          GARY1  L      95
GARY2         83          GARY3   P    188          PRNTR A       LP     66
STRING F       1          XY     ML    256    64


GRO
 NAME     LENGTH RECORD   NAME     LENGTH RECORD   NAME     LENGTH RECORD
B              30          B1     C     128          B2     F     128
BLOCK2 F      128          CALC   M    4000          MBLOCK FL   1655
SP1            22


CATALOG - UFILE
 NAME     LENGTH RECORD   NAME     LENGTH RECORD   NAME     LENGTH RECORD
UFILE  FU    144          WAVES   L     13          YNOT          LP1     66
ZZZZZZ         2
```

PROGRAMMING WITH ARRAYS

SECTION III

WHAT ARE ARRAYS?

An array is a set of variables (or *elements*) which is known by one name. The individual *elements* of an array are specified by the addition of a subscript to the array name. For example, M(7) is the seventh *element* of array M.

REFERENCING ARRAYS

Arrays are referenced by an array variable. The variable can be any single alphabetic character (A through Z). Therefore, you may use up to 26 arrays in a program. The maximum number of data elements allowed in a single array is 5000.

REFERENCING ARRAY ELEMENTS

Arrays have either *one* or *two* dimensions. A *one-dimensional* array consists of a single column of many rows. Individual elements are specified by a single subscript, indicating the row desired. Rows and columns are numbered starting with 1. A *two-dimensional* array consists of a specified number of rows and a specified number of columns organized into a table. For example, an array M of five rows and three columns can be represented as follows:

		Columns		
		1	**2**	**3**
	1	M(1,1)	M(1,2)	M(1,3)
	2	M(2,1)	M(2,2)	M(2,3)
Rows	**3**	M(3,1)	M(3,2)	M(3,3)
	4	M(4,1)	M(4,2)	M(4,3)
	5	M(5,1)	M(5,2)	M(5,3)

Each element of the array is specified by a pair of subscripts separated by commas; the first indicates the row and the second the column.

The remainder of this section deals with the use of arrays. The manipulation of individual array elements is not discussed here since they are treated in the same manner as other numeric variables.

DIMENSIONING ARRAYS

Arrays whose number of rows or columns exceed 10 must be dimensioned using a DIM or COM statement. This allows the system to allocate storage space for the array elements. Arrays whose rows or columns do not exceed 10 do not have to be dimensioned. The system will automatically allocate space (10 elements for one-dimensional arrays, 100 elements for two-dimensional arrays) for undimensioned arrays.

The *physical size* of an array is the total number of elements originally allocated to it. The *logical size* of the array is the current number of rows times the current number of columns. Current row and column numbers can be changed during program execution (refer to RE-DIMENSIONING ARRAYS).

The DIM and COM statements are used to dimension arrays and to set upper bounds on the number of array elements. Arrays and strings can occur in the same dimension statement.

Examples:

```
10 DIM A(10,20),B(30),A$(25),R(30,30)
20 COM D$(255), F(40,10),J(90)
```

REDIMENSIONING ARRAYS

While the *physical size* of an array cannot be changed during execution, the *logical size* can be changed by entering a *new dimensions* parameter with some statements. Those statements allowing you to enter new array dimensions are MAT INPUT, MAT READ, MAT . . . IDN, MAT . . . ZER, and MAT . . . CON. For example, an array with 6 rows and 5 columns (30 elements) could be redimensioned to 10 rows and 3 columns, or to any size so long as the total number of elements does not exceed the *physical size* of the array.

```
10 DIM A(30,20)
 .
 .
 .
20 MAT READ A(600,1)
30 MAT A = ZER(15,40)
```

PLACING VALUES INTO ARRAYS

The values of array elements are undefined when a program begins execution. If you attempt to reference an undefined array element your program will terminate.

There are several methods of assigning values to arrays. Individual elements can be assigned using the assignment statement:

```
10 LET A [ 5 ]=26
20 B [ 1,9 ]=N*4.5
```

In addition, individual elements can appear in INPUT and READ statements.

```
10 INPUT A [ 1 ],A [ 2 ],A [ 3 ]
20 READ B [ 12 ]
```

The MAT assignment statement copies one array into another. The array on the right is copied into the array on the left. The destination array must have as many elements as the source array and the same number of dimensions.

```
10  DIM A[2,3],B[2,3]
20  MAT  READ B
30  MAT A=B
40  MAT  PRINT A
50  DATA 2.5,46.7,75,0,50.1,0,0,0,19.8,0
60  END
RUN

 2.5            46.7           75

 0              50.1           0

DONE
```

The MAT READ statement assigns values from DATA statements to entire arrays, row by row. If the *new dimensions* parameter is specified, the array is given new logical dimensions. The MAT INPUT statement is identical to MAT READ except that the values are entered from your terminal as in an INPUT statement.

If an array is redimensioned in MAT INPUT or MAT READ, the new logical size (i.e., the total number of elements) must not be greater than the size originally allocated to the array, nor may the number of dimensions be altered.

Examples:

In the following example, three elements from each array are printed. Array B is redimensioned by MAT READ in line 30. Note that the DATA statement has the same function with MAT READ as it does with READ.

```
10  DIM A[8],B[10,4]
20  MAT  READ A
30  MAT  READ B[8,4]
40  PRINT A[1],A[5],A[8]
50  PRINT B[1,1],B[5,2],B[8,4]
60  DATA 1,2,3,4,5,6,7,8,9,10
70  DATA 10,9,8,7,6,5,4,3,2,1
80  DATA 30,31,32,33,34,35,36,37,38,39
90  DATA 40,41,42,43,44,45,46,47,48,49
100  END
RUN

 1             5             8
 9             35            49

DONE
```

In the next example, the MAT INPUT statement expects input from the user. Both arrays A and C are printed in their entirety using FOR loops.

(user input is underlined)

```
10  DIM A[3],C[3,2]
20  MAT  INPUT A
30  FOR N=1 TO 3
40  PRINT A[N]
50  NEXT N
60  MAT  INPUT C
70  FOR M=1 TO 3
80  FOR N=1 TO 2
90  PRINT C[M,N]
100  NEXT N
110  NEXT M
120  END
RUN

?27,33,56
 27
 33
 56
?11,22,33,44,55,66
 11
 22
 33
 44
 55
 66

DONE
```

INITIALIZING ARRAYS

Three special functions (MAT . . . ZER, MAT . . . CON, and MAT . . . IDN) provide the means to initialize arrays with certain values, and, optionally, to redimension the arrays.

MAT . . . ZER sets all elements of the array to zero. MAT . . . CON sets all elements of the array to one. MAT . . . IDN assigns an identity array to the array specified. The identity array is all zeros, except the major diagonal (top left corner to bottom right corner), which is all ones.

If an array is redimensioned by MAT . . . ZER, MAT . . . CON, or MAT . . . IDN, the new size cannot have more elements than the physical size, nor can the number of dimensions be altered.

Examples:

Function MAT . . . ZER sets each element of array A to zero.

```
10  DIM A[4,3]
20  MAT A=ZER
30  MAT  PRINT A
40  END
RUN

 0             0             0

 0             0             0

 0             0             0

 0             0             0

DONE
```

MAT A = CON(3,4) redimensions array A to have 3 rows and 4 columns, and sets each element in the newly-dimensioned array to 1.

```
10  DIM A[4,4]
20  MAT A=CON[3,4]
30  MAT  PRINT A
40  END
RUN

 1             1             1             1

 1             1             1             1

 1             1             1             1

DONE
```

MAT A = IDN(4,4) changes the dimensions of A to 4 rows by 4 columns and sets the major diagonal to 1, the remaining elements to 0.

```
10  DIM A[5,5]
20  MAT A=IDN[4,4]
30  MAT  PRINT A
40  END
RUN

 1            0            0            0

 0            1            0            0

 0            0            1            0

 0            0            0            1

DONE
```

PRINTING DATA FROM ARRAYS

The methods of printing data from arrays are parallel to those used for filling arrays. Individual elements can be printed using PRINT:

```
10  DIM A[2,3]
20  MAT  READ A
30  DATA 3.3,46,75,0,50,17
40  PRINT A[1,3]
50  FOR I=1 TO 2
60  FOR J=1 TO 3
70  PRINT A[I,J]
80  NEXT J
90  NEXT I
100  END
RUN

 75
 3.3
 46
 75
 0
 50
 17

DONE
```

MAT PRINT STATEMENT

The MAT PRINT statement allows the printing of one or more complete arrays in a single statement. The elements are printed row by row and can be spaced out in fields or packed together, as in the PRINT statement.

Each row of each array is printed separately, with double spacing between rows. If a comma follows the array, each element starts in one of the five divisions of the line. If a semicolon follows the array, the elements are printed packed together, as if each element were followed by a semicolon. If nothing follows the last array in the statement, a comma is assumed. All formatting is done according to the specifications under the PRINT statement.

Examples:

```
10  DIM A[3],B[5,5],C[2,2]
20  MAT  READ A,B[3,5],C
30  MAT  PRINT A
40  MAT  PRINT B,C
60  MAT  PRINT A;B;
70  DATA 2,4,7,0,5,0,0,0,19,0
80  DATA 1,2,3,4,5,6,7,8,9,10
90  DATA 4.4,3.3
100  END
RUN
 2
 4
 7
 0             5             0             0             0
 19            0             1             2             3
 4             5             6             7             8
 9             10
 4.4           3.3
 2
 4
 7
 0      5      0      0      0
 19     0      1      2      3
 4      5      6      7      8

DONE
```

The MAT READ statement in line 20 redimensions array B. Redimensioning is not permitted in a MAT PRINT statement. Note the effect of the semicolons following A and B in the MAT PRINT statement, line 60, on the printed output.

ARRAY OPERATIONS

The following group of five statements provides functions which operate on one or more entire arrays:

MAT Addition and Subtraction statement
MAT Multiplication statement
MAT . . . INV (Inverse) statement
MAT . . . TRN (Transpose) statement
MAT Scalar Multiplication statement

ARRAY ADDITION/SUBTRACTION

The MAT addition and subtraction statement performs array addition or subtraction (element by element) upon arrays of identical dimensions and assigns the result to another array.

Examples:

```
10  DIM A[2,2],B[2,2],C[2,2]
20  MAT  READ A,B
30  MAT C=A+B
40  MAT  PRINT A,LIN(2),B,LIN(2),C
50  DATA 3,3,4,4,2,2,1,1
60  END
RUN

 3              3
 4              4

 2              2
 1              1

 5              5
 5              5

DONE
```

The values in arrays A and B are added to produce the values printed for array C. Using the same data, A is subtracted from B to produce the following results in C:

```
10  DIM A[2,2],B[2,2],C[2,2]
20  MAT  READ A,B
30  MAT C=A-B
40  MAT  PRINT A,LIN(2),B,LIN(2),C
50  DATA 3,3,4,4,2,2,1,1
60  END
RUN

 3              3

 4              4


 2              2

 1              1


 1              1

 3              3

DONE
```

ARRAY MULTIPLICATION

The MAT Multiply statement performs an array multiplication on an array of dimension m by n and an array of dimension n by p; that is, the number of columns in the first array must equal the number of rows in the second. The result, a new array of dimension m by p, is assigned to a third array.

Each row of the array to the left of * is multiplied by each column of the array on the right to produce the new element. The resulting array is assigned to the array to the left of the assignment operator. This array is redimensioned to dimension m by p. Any or all of these arrays may be the same array.

Examples:

```
10  DIM A[2,3],B[3,2],C[2,2]
20  MAT  READ A,B
30  MAT C=A*B
40  MAT  PRINT A;LIN(1);B;LIN(1);C;
50  DATA 1,2,3,4,5,6
60  DATA 4,5,6,7,8,9
70  END
RUN

 1      2      3
                        } array 1
 4      5      6

 4      5
                        } array B
 6      7
                        } array B
 8      9

 40     46
                        } array C = A*B
 94     109

DONE
```

The method for performing a matrix multiplication is to multiply each element of the first row of array A by the corresponding element of the first column of B and to add the products. The result is the element C(1,1). Then each element in the first row of A is multiplied by the corresponding element in the second column of B and these are added to produce C(1,2). C(2,1) is the sum of the products resulting from the multiplication of row 2 of A and column 1 of B; C(2,2) is the sum of the products of row 2 of A and column 2 of B. To illustrate:

$1\times4\ (4) + 2\times6\ (12) + 3\times8\ (24) = 40$ $\qquad$ $1\times5\ (5) + 2\times7\ (14) + 3\times9\ (27) = 46$

$4\times4\ (16) + 5\times6\ (30) + 6\times8\ (48) = 94$ $\qquad$ $4\times5\ (20) + 5\times7\ (35) \times 6\times9\ (54) = 109$

A second example multiplies the square array C by itself. In this case, the number of columns always equals the numbers of rows.

(user input is underlined)

```
10  DIM C[3,3],D[3,3]
20  MAT  INPUT C
30  MAT  PRINT C;
40  MAT D=C*C
50  MAT  PRINT D;
60  END
RUN

?2,4,6,8,1,3,5,7,9
 2        4        6

 8        1        3

 5        7        9

 66       54       78

 39       54       78

 111      90       132

DONE
```

To achieve the result MAT D=C*C;

D(1,1) = 2×2 (4) + 4×8 (32) + 6×5 (30) = 66
D(1,2) = 2×4 (8) + 4×1 (4) + 6×7 (42) = 54
D(1,3) = 2×6 (12) + 4×3 (12) + 6×9 (54) = 78
D(2,1) = 8×2 (16) + 1×8 (8) + 3×5 (15) = 39
D(2,2) = 8×4 (32) + 1×1 (1) + 3×7 (21) = 54
D(2,3) = 8×6 (48) + 1×3 (3) + 3×9 (27) = 78
D(3,1) = 5×2 (10) + 7×8 (56) + 9×5 (45) = 111
D(3,2) = 5×4 (20) + 7×1 (7) + 9×7 (63) = 90
D(3,3) = 5×6 (30) + 7×3 (21) + 9×9 (81) = 132

This next example multiplies a two-dimensional array with three rows and two columns by a one-dimensional array with two rows. The result is a one-dimensional array with three rows.

```
10  DIM A[3,2],B[2],C[3]
20  MAT  READ A
30  MAT  READ B
40  MAT C=A*B
50  DATA 1,2,3,4,5,6,1,2
60  MAT  PRINT A;LIN(1);B;LIN(1);C;
70  END
RUN

 1       2

 3       4

 5       6


 1

 2


 5

 11

 17

DONE
```

To achieve the result MAT C=A*B:

C(1) = 1×1 (1) + 2×2 (4) = 5
C(2) = 3×1 (3) + 4×2 (8) = 11
C(3) = 5×1 (5) + 6×2 (12) = 17

ARRAY INVERSION

The MAT . . . INV statement assigns the inverse of a square array (i.e., number of rows equals number of columns) to another array. The inverse of an array is the array which, when multiplied by the original array, results in the identity array.

The two arrays must be of the same dimensions. The same array may be used on both sides of the equation.

Example:

(user input is underlined)

```
10  DIM A[5,5],B[5,5]
20  MAT  INPUT B
30  MAT A=INV(B)
40  MAT  PRINT B,LIN(2),A
50  END
RUN

? 1,0,0,0,0,2,1,0,0,0,3,2,1,0,0,4,3,2,1,0,5,4,3,2,1
 1              0              0              0              0

 2              1              0              0              0

 3              2              1              0              0

 4              3              2              1              0

 5              4              3              2              1


 1              0              0              0              0

-2.             1.             1.78814E-07    0              0

 1.            -2.             1.             0              0

 7.74860E-07    .999999       -2.             1              0

-1.01328E-06    7.15256E-07    1.            -2              1

DONE
```

25 values are input to the square array B, then using INV, array A is set to the inverse of B.

ARRAY TRANSPOSITION

The MAT . . . TRN statement assigns the transposition of an *n* by *m* array to an *m* by *n* array. Transposition switches rows and columns.

The array on the left is redimensioned such that the resulting array is the reverse of the original array. The same array cannot be used on both sides of the equation.

Example:

(user input is underlined)

```
10  DIM A[5,3],B[3,5]
20  MAT  INPUT B
30  MAT A=TRN(B)
40  MAT  PRINT B,LIN(2),A
50  END
RUN

? 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
 1             2             3             4             5

 6             7             8             9             10

 11            12            13            14            15


 1             6             11

 2             7             12

 3             8             13

 4             9             14

 5             10            15

DONE
```

Array A is the result of transposing array B with the TRN function. The columns in B are the rows in A; the rows in B are the columns in A.

ARRAY SCALAR MULTIPLICATION

The MAT scalar multiplication statement multiplies all of the elements of an array by a specified value and assigns the result to another array.

Example:

(user input is underlined)

```
10  N=5
20  MAT  INPUT B
30  MAT A=(N*2)*B
40  MAT  PRINT A;
50  DIM A[3,4],B[3,4]
60  END
RUN

?1,2,3,4,5,6,7,8,9,10,11,12
 10     20     30     40

 50     60     70     80

 90     100    110    120

DONE
```

Scalar multiplication simply multiplies each element of the array by the specified *numeric expression*, in this case N*2 or 10 since N=5. Each element of the resulting array A is 10 times the corresponding element in B. The dimensions of A are copied from B. The *numeric expression* must be enclosed in parentheses.

PROGRAMMING WITH STRINGS

SECTION IV

WHAT ARE STRINGS?

Strings are groups of characters which can be used to provide program dialog with the user to facilitate data input or to print comments and headings during report generation. Strings are the principal type of data used in applications such as text editing and most data base applications. For example, the following statement could be used to print a heading in a report generation program:

```
10 PRINT "QUARTERLY REPORT"
```

string

STRING CHARACTER SET

The characters used in strings must be from the ASCII (American Standard Code for Information Interchange) character set. The full set of characters available is given in Appendix A. The character set includes upper and lower case letters, numbers, punctuation, and a variety of non-printing characters (used to control input/output devices). This means that commas, periods, and control characters such as *return* and *line feed* are valid string characters. Strings are normally enclosed in quotes ("THIS IS A STRING"). Examples of strings are:

```
"ABCDEFGHI"
"12345"
"Bob and Tom"
"March 15, 1970"
" "        (Quotes without an enclosed character define the 'null' string)
```

NUMERIC EQUIVALENTS OF CHARACTERS (AN ALTERNATE FORM)

Strings can also be represented as the numeric equivalent of a character preceded by an apostrophe. The numeric equivalent can be any integer from 0 to 255. A complete list of numeric equivalents of characters is given in Appendix A. The numeric equivalent of "A" for example, is 65. The character "A" can therefore be represented as '65. In a similar manner, a *line feed* is '10. Numeric equivalents exist for all ASCII characters including quotes, H^c, and X^c.

Examples:

'23 '64 '49	is equivalent to "W^c@1"
'65 '66 '67	is equivalent to "ABC"

The numeric equivalent form of string character can be combined with other string characters to form longer strings.

Example:

'65 "BC" '68 '69 "F"	is equivalent to "ABCDEF"

When assigning values to strings, two quoted strings cannot be adjacent ("ABCD" '69 is valid, "AB" "CD" '69 is not). The length of the combined string formed cannot exceed the string character limit of 255. Note that the numeric equivalent form of string characters cannot be used in input statements (INPUT, LINPUT, LINPUT#, ENTER, or READ#) or in IMAGE or FILES statements.

UPPER AND LOWER CASE LETTERS

Lower case letters can be input from or output to terminals having a lower case capability. When lower case letters are output to a terminal not capable of printing them, the terminal normally prints the upper case equivalent. Lower case letters are automatically converted to upper case by the system, except when they occur in strings or REM, IMAGE, or FILES statements. Lower case letters in file names used in ASSIGN, CREATE, FILES, and PURGE operations or program names used in CHAIN statements or the exponential "e" in CONVERT statements are converted to upper case when used.

HOW DO YOU REFERENCE STRINGS?

Strings can be used directly as in the statement 10 PRINT "ENTER DATA", or they can be given a name and then the name can be referenced whenever the string of characters is required.

NAMING STRINGS

String names (variables) are made up of a single alphabetic character (A through Z) followed by $, 0$, or 1$. This allows up to 78 different string variables to be defined.

Examples:

```
A$, B0$, C1$
```

DIMENSIONING STRINGS

String variables are assumed to be one character long by default. If a string will exceed one character it must appear in a DIM or COM statement with its maximum length. This ensures that sufficient program space is allocated. Attempts to create a string longer than its dimensioned value will result in an error. Strings of less than the dimensional length are valid.

When a string variable is declared, its "physical" length is set. The "physical" length is the maximum size string that the variable can accommodate.

Examples:

```
710 DIM A$(72),B$(20),C$(50),Z1$(255)
720 COM R$(200),S0$(10)
```

During execution of a program, the "logical" length of a string variable may vary. The "logical" length of the variable is the actual number of characters that the string variable contains at any point.

Examples:

```
100 DIM A$(72)     (Sets physical length of A$ to 72)
200 A$ = "SAMPLE STRING"     (Logical length of A$ is 13)
300 A$ = "LONGER SAMPLE STRING"     (Logical length of A$ is now 20)
```

SUBSTRINGS

Substrings are contiguous subsets of string characters. The subset is defined with subscripts following the string variable name. Two subscripts, separated by a comma, specify the first and last characters of the substring. Characters within a string are numbered from the left starting with one. Subscripts must be positive, non-zero and less than 32,768. (Note that even though subscripts up to 32,768 are allowed, execution errors result when subscripts exceed the string size defined in a DIM statement or 255 characters.) Non-integer subscripts are rounded to the nearest integer.

Example:

```
100 Z$ = "ABCDEFGH"
200 PRINT Z$(2,6)
```

prints the substring

```
BCDEF
```

A single subscript specifies the first character of the substring and implies that all characters following are part of the substring. Continuing the example:

```
300 PRINT Z$(3)
```

prints the substring

```
CDEFGH
```

Two equal subscripts specify a single character substring.

```
400 PRINT Z$(2,2)
```

prints the substring

```
B
```

If a substring is specified which is larger than the physical length of the original string, blanks are appended to the substring in place of the missing characters.

HOW DO YOU USE STRINGS?

Strings may be used in a variety of ways. The first step is to assign a value to the string. Once this has been done there are a variety of operators and string functions that can be used to test or modify the string. When string modification is complete, the string can be output in the same manner as numeric data.

PLACING VALUES IN STRINGS

You assign values to string variables by setting them equal to string data, other strings, or string valued functions.

SIMPLE ASSIGNMENT. The simplest way to set a string value is to use the LET statement. Strings can be set equal to literal strings, other string variables, or substrings.

Examples:

```
10 LET X0$ = "THIS IS A STRING"
20 R$(1,5) = G$(6,10)
```

In the first example, X0$ is assigned the value in quotes. In the second example, the value contained in the substring G$(6,10) is assigned to the substring R$(1,5).

STRING DATA. String variables can be set equal to string data using the INPUT, LINPUT, ENTER, or READ statements. The first three statements are used to set a string or substring equal to string data entered from your terminal.

Examples:

```
10 INPUT A$                (Inputs a string value from your terminal and assigns it to A$)
20 INPUT B0$, C$(1,5)      (Inputs a string value and assigns it to B0$ and inputs a second
                            string value and assigns it to the first five characters in C$)
30 LINPUT X$               (Inputs an entire line from your terminal and assigns it to X$)
40 ENTER #V, E, A, A$      (Inputs a string value from your terminal and assigns it to A$)
```

The READ statement can be used to set a string or substring equal to string data contained in a DATA statement. Strings used in DATA statements are limited to a maximum of 255 characters. (This may be less depending on your system's configuration.)

Example:

```
10 READ A$,B$,C$,N,D$(6,8)
20 DATA "ABC","BCA","CAB",5,"BA" '67
```

The example assigns eacl of the values given in the DATA statement to the corresponding variable in the READ statement. Note that string and numeric variables can occur in the same READ statement.

The READ# and LINPUT# statements can be used to obtain values from a file rather than your terminal. (Refer to Section V for a discussion of file operations.)

Examples:

```
10 READ# 1;A$,B$,C$(1,7)
20 LINPUT# N; G$
```

SETTING STRINGS EQUAL TO STRING VALUED FUNCTIONS. String valued functions (refer to the discussion following) can be used in place of string data in assignment operations. The string variable is set equal to the result of the function.

Examples:

```
10 A$ = UPS$(B$)
20 Q$(3,3) = CHR$(N)
```

USING STRINGS IN RELATIONAL OPERATIONS

Strings and string variables can be used with relational operators in the same manner as numeric values. The relational operators (=, <, >, >=, <=, < >) can be used with strings to perform branching operations. These operators are also useful for sorting strings.

Example:

```
10 IF A$ >= B$(3,7) THEN 40
20 IF C$ = "ABC" THEN 50
```

STRING STATEMENTS AND FUNCTIONS

There are several statements and functions that perform special string operations. Some use a string or numeric argument and return string values, others return numeric values.

STRING VALUED STATEMENTS AND FUNCTIONS. The statements and functions that return string values are the CONVERT statement and the CHR$, and UPS$ functions.

The CONVERT statement allows you to change numeric data to its ASCII equivalent and an ASCII number to its numeric equivalent. (Note, this is not the same as the equivalents of individual characters as in CHR$ or NUM.) In the following example the value of N is converted to the string of ASCII characters that would be listed (LIST command) at your terminal by the statement PRINT N.

Example:

```
10 N = 2*(111)
20 CONVERT N TO A$
```

A$ now contains the ASCII characters "2", "2", and "2". The ASCII string representation can now be used in alphanumeric sort operations or can be appended to other strings.

For example, if N were used as a counter for labeling files, a new file name (string) could be created by appending the counter value (number) each time a new file is required.

Example:

```
 10 F$ = "FIL"
 20 N = 0
 30 GOSUB 100
  .
  .
  .
 90 REM FILE CREATOR
100 N = N + 1
110 CONVERT N TO A$
120 F$(4) = A$
130 CREATE R, F$, 100
140 RETURN
```

The first time the subroutine is entered, a file name FIL1 containing 100 records would be created. Additional passes through the subroutine would create the files FIL2, FIL3, FIL4, etc.

The CHR$ function converts a number in the range $0 \leq n \leq 255$ into the equivalent ASCII character. (This is the opposite of the NUM function which returns a numeric equivalent for a string character.) A complete list of ASCII characters and their numeric equivalents is given in Appendix A.

Example:

```
10 FOR N = 1 to 255
20 A$(N) = CHR$(N)
30 NEXT N
```

This example would create a string (255 characters long) where each character is the ASCII equivalent of its position in the string. In other words A$(65) = "A", A$(66) = "B", A$(67) = "C", etc.

The UPS$ function causes lowercase alphabetic characters (a through z) to be shifted to their uppercase equivalents (A through Z).

Example:

```
10 A$ = UPS$(Z$)
```

The UPS$ function is valuable in alphanumeric sorts. Sorting is normally done according to the ASCII equivalent value of each character. If the strings to be sorted contain a mix of upper and lowercase characters it is possible that an "a" would be set after "Z". To eliminate this problem you can shift strings to their uppercase equivalent before sorting. Once the sort position is obtained the lowercase string can be entered in the proper table position.

NUMERIC VALUED STATEMENT AND FUNCTIONS. The statements and functions that use string arguments and return numeric values are the CONVERT statement and the LEN, NUM, and POS functions.

The CONVERT statement allows you to change an ASCII number to its numeric data equivalents and numeric data to its equivalent in ASCII characters. (Note, this is not the same as the equivalents of individual characters as in CHR$ and NUM.) In the following example the value of A$ is converted to a numeric data value and assigned to N. This is just the opposite of the example used earlier with the CONVERT statement.

Example:

```
10 A$ = "995.6"
20 CONVERT A$ TO N
```

The numeric variable N now contains the numeric value 995.6. This numeric value can now be used in numeric expressions. This conversion is useful when formatting reports involving arithmetic operations and textual data.

The LEN function returns a numeric value that is the length (number of characters) currently assigned to a string variable. In the example that follows, N is set to the number of characters assigned to (not dimensioned for) A$.

Example:

```
10 DIM A$(72)
20 A$ = "ABC"
30 N = LEN(A$)
```

This would result in N being set to 3. The LEN function is useful in sorting or searching strings or in appending or modifying strings. The example given for the NUM function illustrates how the LEN function can be used to index character by character through a string.

The NUM function converts a single string character to a numeric value that is its ASCII code equivalent. A complete list of ASCII characters and their numeric code equivalent is given in Appendix A. In the following example, the numeric variable N is set to the ASCII code equivalent of the third character in A$.

Example:

```
10 N = NUM (A$(3,3))
```

You can obtain the numeric code equivalent of a whole string as follows:

```
10 FOR I = 1 TO LEN (A$)
20 N(I) = NUM (A$(I,I))
30 NEXT I
```

The POS function allows you to test to see if one string occurs within another string. This is a powerful tool in any text editing or other word processing application. The POS function uses two string arguments, a string to search and a string to look for. If the "looked for" string is found, the function is set to the position in the "searched" string where the first matched character occurs. If the "looked for" string does not occur, the function is set to 0. In the following example, S$ is a long string that may contain one or more occurrences of another string R$.

Example:

```
10  DIM S$[60],R$[10]
20  S$="THIS IS THE POINT TO INPUT THE DATA"
25  R$=" THE "
30  REM P IS THE POSITION IN S$ WHERE R$ BEGINS
35  P=0
40  P1=POS(S$[P+1],R$)
50  IF P1=0 THEN 80
55  P=P1+P
60  PRINT "SEARCH STRING FOUND AT";P
70  GOTO 40
80  END


RUN

SEARCH STRING FOUND AT 8
SEARCH STRING FOUND AT 27

DONE
```

OUTPUTTING STRINGS

Strings can be printed at your terminal, written to disc files, or output to any of the peripheral devices available on your system. Output of string data is accomplished in the same manner as numeric data. String and numeric data can be mixed in the same output statement. Output is performed using the PRINT and PRINT# statements.

The PRINT statement allows you to output string data to your terminal.

Example:

```
100 PRINT "THIS IS A STRING"
120 PRINT A$; B$, C$
```

The PRINT USING statement can be used to output string data according to a predetermined format. In the following example, the variables A$, I, and T(I) are printed using the format defined in statement 80. (Quoted strings and string valued functions cannot be used as print list items in a PRINT USING statement.)

Example:

```
10  DIM A$[10],B$[20]
20  A$="ACCOUNT"
30  B$="TOTAL REPORT"
40  PRINT TAB(10),B$[7,12]
45  T=0
50  FOR I=1 TO 10
60  READ T[I]
70  PRINT  USING 80;A$,I,T[I]
80  IMAGE 7A,DD,5X,6D.D
85  T=T+T[I]
90  NEXT I
95  PRINT
100  PRINT  USING 105;B$[1,5],T
105  IMAGE 2X,5A,7X,6D.D
110  DATA 15,150,2000,3000,600,50,70,19.5,100,120
120  END
RUN
OK


          REPORT
ACCOUNT 1          15.0
ACCOUNT 2         150.0
ACCOUNT 3        2000.0
ACCOUNT 4        3000.0
ACCOUNT 5         600.0
ACCOUNT 6          50.0
ACCOUNT 7          70.0
ACCOUNT 8          19.5
ACCOUNT 9         100.0
ACCOUNT10         120.0

  TOTAL          6124.5
```

Refer to Section XI for further information on the PRINT USING and IMAGE statements.

The PRINT# statement can be used to output data to BASIC formatted or ASCII files (refer to Section V for a discussion of file operation).

Example:

```
70 PRINT#1;A$,N,B$,"OUTPUT"
80 PRINT#N,R+1;A$,B$(1,7),H(3),"TEST"
```

FILES

SECTION V

There are two types of files provided by the system: BASIC formatted files and ASCII files.

BASIC FORMATTED FILES

BASIC formatted files are used primarily to store data for later retrieval and manipulation. You have already seen how data can be stored within a program using DATA statements. For example, the following program prints out the months of the year by reading them from two data statements:

```
10  DIM A$[40]
20  PRINT "THE MONTHS OF THE YEAR:"
30  FOR I=1 TO 12
40  READ A$
50  PRINT A$
60  NEXT I
70  DATA "JANUARY","FEBRUARY","MARCH","APRIL","MAY","JUNE","JULY"
80  DATA "AUGUST","SEPTEMBER","OCTOBER","NOVEMBER","DECEMBER"
90  END
```

BASIC formatted files are similar to DATA statements. For example, each can store both numeric and string data; each has associated with it a pointer which is initially set to the first item of the first data stetement and which advances sequentially through the data as items are read; each can be read serially (with the READ and READ#; statements, respectively); each can be read directly with the RESTORE and READ#; statements, respectively); and each can detect (with the TYP function) whether the next item to be read is a number, a string, or if there is more data to be read.

DATA statements, although useful in many cases, are not very powerful for two reasons: (1) The amount of data which can be stored in any one program is limited to the maximum size of a program (about ten thousand words of storage, which is the equivalent of about twenty thousand characters of data). (2) The data stored in DATA statements never changes from one RUN of the program to the next.

BASIC formatted files fill these deficiencies. They may be used to store large amounts of data (a BASIC file may contain up to 16.7 million characters of data, depending on the system's configuration); and, this data may be added to, changed, and deleted by running programs.

CREATING AND PURGING A BASIC FORMATTED FILE

Before you can store data in a BASIC formatted file, space for that file must be reserved on a disc using either the CREATE statement or the CREATE command. Each permits you to specify the three defining characteristics of a file:

name	— 1 to 6 alphanumeric characters and unique from all other names in your library
length in records	— can be up to 32,767 records long depending on system configuration; automatically numbered consecutively from record 1
size of each record (optional)	— all records physically occupy 256 *words* of system storage although for special programming purposes they may be set to a logical length from 64 to 256 *words* (default = 256 *words*)

For example, to create a file to store a list of 1000 employee names with their ID numbers and salaries, use the command

```
CRE-SALARY,59
```

Since no value is indicated for record size, the default value, 256 *words*, is assumed.

To determine how long the file needs to be, you must first approximate how many *words* of record space each of the 1000 entries will need. Every *number* in a file occupies 2 *words*; every *string* of n characters occupies $(n/2 + 1)$ *words* — 2 characters per *word* plus 1 *word* for the length of the string. A string with an odd number of characters uses one *word* for the last character. Each entry in the file SALARY will be of the form: ID, name, salary. Allowing for a maximum length of 20 characters per employee name, each entry will occupy at most 15 *words* of space in a record (2 *words* for the ID number, 2 *words* for the salary, and 11 *words* for the 20 character name string). Each record (by default) can store 256 *words*. Data items cannot be split up (putting part of an item at the end of one record and the rest of the item at the beginning of the next record). Therefore, each record can hold 17 data items:

256 *words* ÷ 15 *words/item* = 17 *items*, with 1 *word* left over

Since the file SALARY is to hold data for 1000 employees and 17 data items fit in a record, 59 records should be reserved for the file:

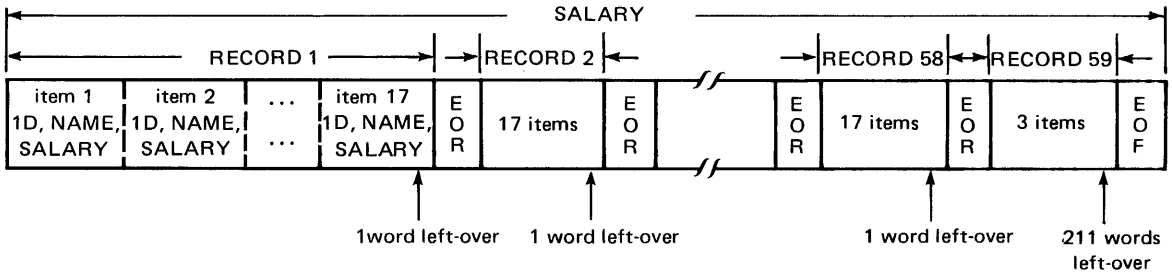
1000 *items* ÷ 17 *items/record* = 59 records

It is possible that the data will actually occupy less than 59 records of storage. If an employee name is shorter than 20 characters then that data item will occupy less than 15 words in the record. It may be possible to fit more than 17 data items in a record, hence 1000 data items might take fewer than 59 records. However, you should still reserve enough space for the largest possible case — 1000 items where every name is 20 characters long. Also, if you do not presently have 1000 employees for the file, but anticipate eventually adding new employees, it would be wise, when you initially create the file, to reserve sufficient space to extend the list.

When a file is first created and before any data is entered in the file, an End-Of-File (EOF) mark is automatically written in the first word of each record. When data is entered in a record it is written over the EOF mark, eliminating it. You can place EOF marks in a record using the PRINT statement as will be seen in example 1.

Once created, a file remains on a disc of the system until it is eliminated with a PURGE statement or command.

This is how the file SALARY, with 1000 data items, all of maximum length (15 words), will be arranged on the disc:



Each pair of consecutive records is separated by an "End-Of-Record" (EOR) mark, and the last record of the file is followed by an "End-Of-File" (EOF) mark. Empty (left-over) words within a record are ignored.

OPENING AND CLOSING A BASIC FORMATTED FILE

In order for a program to access a file it must first open the file with a FILES statement or an ASSIGN statement. Opening a file causes the file to be associated with an integer *file number* from 1 to 16. Once a file is opened, all references to it are through its *file number*. For example, if, in a program, 3 files are opened with the statement 10 FILES A,B,C then file A will be referred to as file #1, file B as #2, and file C as #3. If the statement ASSIGN "D",1 appears later in the program, file D will become file #1, effectively closing file A. (Of course, files A, B, C, and D must have been previously created in your library.)

Opening a file also sets its file pointer to the first word of the first record of the file. There is one file pointer for each opened file in a program and it points to where the next data item to be printed in the file will go, or where the next data item to be read from the file is located.

All files in a program are automatically closed upon program termination (if they have not already been closed during program execution with an ASSIGN statement.

The following four examples illustrate the use of most file statements and functions (serial READ, serial PRINT, IF END statement, UPDATE statement, REC and ITM functions, ADVANCE statement) in performing four fundamental types of data manipulations: adding data to a file, changing data within a file, removing data from a file, and listing the contents of a file.

EXAMPLE 1. The following program can be used to add new employees to the file SALARY. It first checks to see if a new entry (ID, name, and salary) is valid — that the ID number is a new integer between 1 and 1000. It then reads each employee entry sequentially checking for a duplicate ID number. If a duplicate is found, an error message is printed, the file pointer is reset to the beginning of the file, and another ID, name, and salary are requested. If the ID is not a duplicate, the new entry will be added to the file immediately following the last entry in the file. The last entry is located by looking for an EOF mark on the record. When the first item is entered in the file, the first EOF mark will be found at the beginning of record 1. (Remember that when a file is created, EOF marks are automatically put at the beginning of every record in the file.)

The statement that prints a new item on the record:

```
150 PRINT #1;I,N$,S,END
```

prints on file #1 (SALARY file) the ID number (I), the employee's name (N$), the salary (S), and an EOF mark (END). The new entry, which is now the last data item in the file, will write over the old EOF mark (eliminating it) and be followed by a new EOF mark.

If the file is full, the message "FILE FULL" is printed and the new entry is not added.

```
10  FILES SALARY
20  DIM N$[20],N1$[20]
30  PRINT "ID,NAME,SALARY";
40  INPUT I,N$,S
50  IF I >= 1 AND I <= 1000 AND I=INT(I) THEN 80
60  PRINT "ILLEGAL ID"
70  GOTO 30
80  IF  END #1 THEN 140
90  READ #1;I1,N1$,S1
100  IF I <> I1 THEN 90
110  PRINT "DUPLICATE ID"
120  READ #1,1
130  GOTO 30
140  IF  END #1 THEN 170
150  PRINT #1;I,N$,S, END
160  STOP
170  PRINT "FILE FULL"
180  END
```

Statement Number	Meaning
10	Specify file to be used.
20	Dimension strings N$ and N1$.
30,40	Ask for and get a new entry.
50	Check to see that ID is an integer between 1 and 1000.
80	Set up IF END condition so that when the file pointer hits an EOF mark, (which will indicate the last entry in the file), the program will branch to statement 140.
90	Read from file #1 an ID(I1), name(N1$), and salary(S1).
100	If the ID from the new entry (statement 40) is the same as the ID just read from the file (statement 90) then you have tried to enter a duplicate ID number. 120 Reset the pointer at the first entry of the SALARY file. 130 Go back to statement 30 and ask for a new entry. If they are not the same, go back to 90 and check the next entry in the file. Continue checking until there are no more entries in the file (the pointer will be at an EOF mark).
140	Reset IF END condition so that if there is not enough space to print the new entry, the message FILE FULL will be printed.
150	Print the new entry, followed by an EOF mark, on file #1. (This new entry will over-write the old EOF mark.)

EXAMPLE 2. The following program can be used to change the value of an employee's salary within the file SALARY. The UPDATE statement (160 UPDATE #1; S) replaces the next sequential data item in file #1 with a new value of S.

```
10  FILES SALARY
20  DIM N$[20]
30  PRINT "ID";
40  INPUT I
50  IF I >= 1 AND I <= 1000 AND I=INT(I) THEN 80
60  PRINT "ILLEGAL ID"
70  GOTO 30
80  IF  END #1 THEN 180
90  READ #1;I1
100  IF I=I1 THEN 130
110  READ #1;N$, S
120  GOTO 90
130  PRINT "NEW SALARY";
140  INPUT S
150  READ #1;N$
160  UPDATE #1; S
170  STOP
180  PRINT "NO SUCH ID"
190  READ #1, I
200  GOTO 30
210  END
```

Statement Number	Meaning
30	Ask for ID number of employee whose salary is to be changed.
80	Set up an IF END condition so that if you read through all the items in the file and do not find the ID, the message NO SUCH ID will be returned.
90	Read an ID from file #1.
100	If it matches the number to be changed (statement 40) then: 130 Ask for the new salary. 150 Read the name (just to move the pointer past the name). 160 Replace the old salary with the new salary. If it does not match, then read past the name and salary for this ID and read the next ID.

EXAMPLE 3. One way to delete an entry from the file is to replace that employee's entry with a line of zeros and blanks. Note that this does not actually remove the entry and leave empty space on the file. It only writes over the entry with zeros and blanks. The following program introduces two functions, REC and ITM, and the ADVANCE statement. The REC function returns the current record number being accessed in the file specified. The ITM function returns the number of data items (numbers and strings) between the beginning of the currently accessed file record and the position of the file pointer for a file. The ADVANCE statement advances the file pointer T items within the record.

```
10  FILES SALARY
20  DIM N$[20]
30  PRINT "ID";
40  INPUT I
50  IF I >= 1 AND I <= 1000 AND I=INT(I) THEN 80
60  PRINT "ILLEGAL ID"
70  GOTO 30
80  IF  END #1 THEN 210
90  R=REC(1)
100  T=ITM(1)
110  READ #1;I1
120  IF I=I1 THEN 150
130  READ #1;N$,S
140  GOTO 90
150  READ #1,R
160  ADVANCE #1;T,X
170  UPDATE #1;0
180  UPDATE #1;""
190  UPDATE #1;0
200  STOP
210  PRINT "NO SUCH ID"
220  READ #1,1
230  GOTO 30
240  END
```

Statement Number	Meaning
80	Set up IF END condition to print the message NO SUCH ID if the ID number entered in statement 30 does not match any ID in the file.
90	Set R to the number of the record that the file pointer is in.
100	Set T to the number of entries between the beginning of the record and the file pointer. R and T thus exactly define the location of the ID number about to be read.
110	Read this ID from the file.
120	See if the ID just read is the one to be deleted. If it is not:
	130 Read through the rest of the entry (name and salary) for that ID. The file pointer is now at the ID number for the next entry.
	140 Return to statement 90 and read the next ID from the file.
	If the ID to be deleted has been found:
	150 Set the file pointer at the beginning of the record in which the matching ID was found (recall statement 90 R=REC(1)).
	160 Advance the file pointer T items from the beginning of record R. It is now pointing to the ID to be deleted.
	170 Replace the old ID number with a zero.
	180 Replace the old name with blanks.
	190 Replace the old salary with a zero.

EXAMPLE 4. To obtain a complete listing of employees' salaries without listing deleted items (i.e. rows of zeros and blanks):

```
10  FILES SALARY
20  DIM N$[20]
30  IF  END #1 THEN 80
40  READ #1;I,N$,S
50  IF I=0 THEN 40
60  PRINT I,N$,S
70  GOTO 40
80  END
```

Statement Number	Meaning
30	Set IF END condition to stop reading at the end of the data.
40	Read an entry from the file.
50	If the entry's ID is zero then that entry was deleted (see example 3) and should therefore not be printed.
60	Print the entry (since its ID is not zero).

The programs prepared thus far to enter, list, and alter entries in the file SALARY have used serial file accessing. The programs read or write a serial list of data items (either numbers or strings of characters) without regard to the underlying structure of the file. The program in example 1 writes data items into the file in serial order. Each write operation begins where the previous one left off. Then, to retrieve one of these items, the program resets the pointer at the beginning of the file and reads through the items until it comes to the desired item. There is only one pointer for each file. When the pointer is positioned by either a READ or a PRINT statement, it remains pointing to the next item in the file until it is repositioned by another file operation. To access a specific data item in a file using serial access, all data located in front of that item in the file must be accessed. If there is a lot of data in the file, serial access can be very slow. With additional programming effort, any BASIC formatted file can be used for direct access storage. In this case, the program breaks the file into a series of subfiles that can be modified independently. A READ or PRINT access of a file is 'direct' if it specifies a particular record within a file.

Serial Access

```
100 READ #1;A,B,C       (reads from the current position of the #1 file pointer)
```

Direct Access

```
100 READ #1,5;A,B,C     (moves the #1 file pointer to record 5 before reading)
```

The system provides statements that take advantage of record structure. The file pointer can be moved directly to the beginning of any record. Also, any record can be read or printed independently of the rest of the file using direct access versions of the READ# and PRINT# statements.

The next four examples perform exactly the same operations as the first four examples in this chapter but use direct file access instead of serial file access. Because we do not need files that contain duplicate information, we can remove the SALARY file from the disc using the PURGE command:

```
PURGE-SALARY
```

Now create a new BASIC formatted file with the same name:

```
CRE-SALARY,1000
```

To use direct access, you must know exactly which entry the record is in. In the following examples, each entry is stored in its own record (i.e. ID number 1 is in record 1, ID #2 is in record 2, etc.). Therefore, to reserve storage space for 1000 employees, the file SALARY must be created with 1000 records. Having purged the old file SALARY, we now CREATE-SALARY,1000 (record size assumes default value of 256 words).

EXAMPLE 5. A new employee entry with ID number I will be printed in the file SALARY at record I. This record can be accessed directly (that is, without first reading through preceding records).

```
10  FILES SALARY
20  DIM N$[20],N1$[20]
30  PRINT "ID,NAME,SALARY";
40  INPUT I,N$,S
50  IF I >= 1 AND I <= 1000 AND I=INT(I) THEN 80
60  PRINT "ILLEGAL ID"
70  GOTO 30
80  READ #1,I
90  IF TYP(1)=3 THEN 120
100  PRINT "DUPLICATE ID"
110  GOTO 30
120  PRINT #1;I,N$,S
130  END
```

Statement Number	Meaning
30,40	Input entry (ID, name, salary).
80	Directly read to record I in file #1.
90	The TYP function determines the type of data item the file pointer is at. If TYP(1) = 3 then the file pointer is at an EOF mark so there is no data in that record.
	120 Enter the new data item.
	If the file pointer is not at an EOF mark, then there is already data in that file.

EXAMPLE 6. The following program uses direct file access to change an employee's salary. In statement 90 the TYP function tests to see if the file pointer is at a number. Since each record contains either an EOF mark or a data item which begins with an ID number, if the pointer is not at a number, there is no data item in that record. Recall that all ID numbers are stored in corresponding record numbers. We want to change the salary for the employee whose ID number is I. The old salary for ID number I will be in record I. If no ID number is found in record I, then no employee has ID number I.

```
10  FILES SALARY
20  DIM N$[20]
30  PRINT "ID";
40  INPUT I
50  IF I >= 1 AND I <= 1000 AND I=INT(I) THEN 80
60  PRINT "ILLEGAL ID"
70  GOTO 30
80  READ #1,I
90  IF TYP(1)=1 THEN 120
100  PRINT "NO SUCH ID"
110  GOTO 30
120  READ #1;I,N$
130  PRINT "SALARY";
140  INPUT S
150  UPDATE #1;S
160  END
```

Statement Number	Meaning
80	Directly read to record I in file #1.
90	If TYP(1) = 1, the next data item is a number.
	120 Read past the ID number and the name.
	130 Input new salary.
	150 Replace the old salary with the new salary.
	If the next data item is not a number, the message NO SUCH ID is returned and the program returns to accept another ID number.

EXAMPLE 7. This direct file access program to delete an employee's data entry from the file is much simpler than the corresponding program using serial file access. Recall that in the serial access example in order to delete an entry, zeros and blanks were written over the old information. With direct file access, the item can actually be wiped out of the file without additional programming by printing an EOF mark in the record.

```
10  FILES SALARY
20  DIM N$[20]
30  PRINT "ID";
40  INPUT I
50  IF I >= 1 AND I <= 1000 AND I=INT(I) THEN 80
60  PRINT "ILLEGAL ID"
70  GOTO 30
80  PRINT #1,I; END
90  END
```

Statement Number	Meaning
80	Print at record I an EOF mark.

EXAMPLE 8. To obtain a listing of file entries in the direct access mode is less efficient than in serial access. In this case every record, from 1 to 1000, must be read. You must programmatically direct a read to each record by incrementing the record number. Serial reads can be accomplished simply by looping (record boundaries are ignored).

```
10  FILES SALARY
20  DIM N$[20]
30  IF  END #1 THEN 70
40  FOR I=1 TO 1000
50  READ #1,I;I,N$,S
60  PRINT I,N$,S
70  NEXT I
80  END
```

Statement Number	Meaning
30	Set IF END condition to go to the next record if an EOF mark is read.
40-70	Read every record from 1 to 1000, printing the contents of any record that contains data. (If a record does not contain data, it contains an EOF mark and is skipped.)

MULTIPLE ACCESS

The information contained in the file SALARY may be read by several programs at the same time. However, only one program at a time may have write access to the file. If more than one program simultaneously attempts to modify SALARY, a WRITE TRIED ON READ-ONLY FILE message is returned to all but the first program. This is because the file SALARY is a Single Write Access file. All files are normally created with Single Write Access (SWA).

It is possible to allow several programs to have concurrent write access to the same file by using the Multiple Write Access (MWA) command. (Refer to Section X for a complete discussion of the MWA command.) For instance, the programs in example 6 (changing employee salaries) and example 7 (deleting file entries) both contain statements that write to the SALARY file. To run the two programs simultaneously you must first specify MWA for the file by entering the command

```
MWA-SALARY
```

The SYSTEM statement (refer to Section XI) can be used within a program to specify MWA:

```
15 SYSTEM N,"MWA-SALARY"
```

This statement will return a value of 0 for N if the MWA command is successful and 1 if it is not.

Now, having entered the MWA command, both programs can simultaneously write to the SALARY file. To avoid confusion when multiple users are modifying the contents of the same MWA file, the LOCK and UNLOCK statements should be used. The LOCK statement is used to test whether or not a file is busy. It does not deny others access to the particular file but it allows users to cooperate in accessing that file. For instance, programs A and B below correspond to the programs seen in examples 6 and 7 with LOCK and UNLOCK statements inserted.

A. Changing Employee Salaries

```
10  FILES SALARY
20  DIM N$[20]
30  PRINT "ID";
40  INPUT I
50  IF I >= 1 AND I <= 1000 AND I=INT(I) THEN 80
60  PRINT "ILLEGAL ID"
70  GOTO 30
80  READ #1,I
90  IF TYP(1)=1 THEN 120
100  PRINT "NO SUCH ID"
110  GOTO 30
120  READ #1;I,N$
130  PRINT "SALARY";
140  INPUT S
145  LOCK #1
150  UPDATE #1;S
155  UNLOCK #1
160  END
```

B. Deleting File Entries

```
10  FILES SALARY
20  DIM N$[20]
30  PRINT "ID";
40  INPUT I
50  IF I >= 1 AND I <= 1000 AND I=INT(I) THEN 75
60  PRINT "ILLEGAL ID"
70  GOTO 30
75  LOCK #1
80  PRINT #1,I; END
85  UNLOCK #1
90  END
```

If program A attempts to write to the file before program B does, program A will be given write access to the file. If program B now attempts to write to the file, execution of program B will pause until program A has unlocked the file. Be sure to match every LOCK statement in a program with an UNLOCK statement. If the UNLOCK statement had been left out of program A, the file would have remained locked and program B could not have written on the file until program A terminated. (Refer to Section XI for a detailed description of the LOCK and UNLOCK statements.)

A file that has been operating in MWA mode can be changed back to the SWA mode using the SWA command:

```
SWA-SALARY
```

READ/WRITE RESTRICTIONS

The restriction option of the ASSIGN statement can be used to provide additional control of read/write access to an SWA file. A read and write restriction can be placed on a file to prevent subsequent users from accessing the file while you are using it. For example, to apply the read and write restriction to the file SALARY you can use the statement:

```
5 ASSIGN "SALARY",1,N,RR
```

If another user has already opened the SALARY file and restricted its use, then the ASSIGN statement will return a value of 6 or 7 for the return variable N and the file will not be opened. (Refer to Section XI for a complete discussion of the ASSIGN statement and its return variables.) Otherwise, the file will be opened and the RR (read/write) restriction set. No other user may access the file SALARY in any way until your program removes the RR restriction by closing the file or ASSIGNing it again with no restriction.

Additional file access information can be found in Section VIII.

ASCII FILES

ASCII files provide the user with programmable access to the following devices: magnetic tape units, paper tape readers, paper tape punches, card readers, and line printers. In addition, ASCII disc files complement BASIC formatted files by providing additional means to store and access information on the system.

CHARACTERISTICS OF ASCII FILES

ASCII files, like BASIC formatted files, consist of a series of consecutive records separated by end-of-record (EOR) marks and followed by an end-of-file (EOF) mark.

ASCII files can hold only string data. Numeric data is automatically changed to a character string that represents the numeric data. For instance, in the example shown below, "N" is set to the numeric value 176.54. ASC1 is an ASCII disc file. The character string " 176.54" would be printed on the file and not the numeric value of the data.

```
10 FILES ASC1
20 N = 176.54
30 PRINT #1; N
40 END
```

Note that only the number 176.54 in a two word format would have been stored in a BASIC formatted file. The ASCII file ended up with seven characters being stored, and the data content of the files is different. A BASIC formatted file would have contained the binary representation of a signed number and an exponent. The ASCII file contains the binary representation of the ASCII characters that would have been sent to the terminal.

ASCII file access is limited in comparison with BASIC formatted files:

- Except for ASCII files resident on magnetic tape or disc, a given file is read-only or write-only. An example of a read-only file is a card reader; a line printer is a write-only file.
- Because of the serial nature of paper tape readers, paper tape punches, card readers, and line printers, data read from or written to ASCII files associated with these devices cannot be re-accessed.
- The ability to selectively alter portions of an ASCII file is severely restricted or non-existent, depending on the device.
- Direct access READs and PRINTs are not allowed.
- For all ASCII file devices multiple user access is prohibited (hence, they are called 'non-sharable devices').
- The ADVANCE, UPDATE and CREATE statements are not permitted with ASCII files.
- The REC and ITM functions cannot be used with ASCII files.

CREATING AND PURGING ASCII FILES

ASCII files are created via the FILE command or the SYSTEM statement. Each includes specification of a *name* (1 to 6 letters or digits) and a *device type*. For ASCII files on disc, the *number of records* must also be specified. Users can be granted access to non-sharable devices by the system operator. Disc-resident ASCII files, like BASIC formatted files, are available to all users. The DEVICE command displays a list of devices available to you.

```
DEV
  DEVICE        MAXIMUM
DESIGNATOR  RECORD SIZE   STATUS
   CRØ            4Ø
   LPØ            66        BUSY
   LP1            66
   PPØ            64
   MTØ           256
   PRØ            64
```

The items in the first column are mnemonics assigned by the system operator that represent the various devices available. For instance, CR0, is card reader number 0; LP0 and LP1 are line printers 0 and 1.

The numbers in the second column indicate the maximum number of words allowed per record for each device. The card reader can hold records up to 40 words long (hence a card of 80 columns — 2 columns/word); the line printers can hold up to 66 words per record; etc.

The status column shows that LP0 is busy. If you were to request that device, your program would terminate.

An N/A entry in the STATUS column would indicate that the system operator had removed that device from the system or had given another user exclusive access to that device. (Refer to Section I for a discussion of system operator capabilities.)

If no *record length* is specified when creating an ASCII file, the maximum record size is assigned by default. For ASCII files on disc, the *number of records* must be designated. (The default value for disc *record length* is 63 words.)

```
FILE — ASC1, CR0
FILE — ASC2, DS, 50
FILE — ASC3, LP1
FILE — ASC4, MT, 80
```

(Note: These ASCII files will be used in later examples in this section.)

ASC1 is an ASCII file equated to the card reader CR0; its record length is 40 words by default. ASC2 is an ASCII disc file with 50 records of 63 words each. ASC3 is a line printer with the default record length 66 words. ASC4 is the magnetic tape unit with record length 80 words.

ASCII files can be purged with either the PURGE command or the PURGE statement.

OPENING ASCII FILES

Once an ASCII file has been created, it can be opened for use in a program by either a FILES statement or an ASSIGN statement.

```
10 FILES ASC1, ASC2, ASC3
```

If the file is disc-resident, the program is given access to it. Otherwise, the associated device is requested. If a device is not available an error indication is returned to you. Opening a file causes the file to be associated with an integer *file number* from 1 to 16, according to its position in the FILES statement. In the example above, ASC1 becomes file #1, ASC2 is file #2, and ASC3 is file #3. Once a file is opened, all references to it are through its file number. The ASSIGN statement can be used in a program to re-assign file numbers. For instance, the following statement will open the file ASC4 and will associate ASC4 with file #2, closing the file ASC2:

```
20 ASSIGN ASC4,2,R
```

PRINTING TO AN ASCII FILE

Data is written to an ASCII file using the file PRINT statement. For example, having created the files ASC1, ASC2, ASC3, and ASC4, open the ASCII file associated with the line printer and print the number 1975 on that file:

```
10 FILES ASC3
20 PRINT #1; 1975
30 END
```

Recall that in ASCII files numeric data is automatically changed to a character string that represents the numeric data. The data is formatted on the ASCII file exactly as it would appear on a terminal. Multiple items in an ASCII file PRINT statement are formatted on the ASCII file according to the format specifications for the PRINT statement. (Refer to Section XI.)

For example, the PRINT statement

```
60 PRINT 1975, "HELLO", -127.5
```

would cause the character string

(*1 blank*) 1975 (*10 blanks*) HELLO (*10 blanks*) -127.5

to be printed as the next line on your terminal.

Correspondingly, for ASCII file PRINT statements, the program

```
10 FILES ASC3
20 PRINT #1; 1975, "HELLO", -127.5
30 END
```

would cause the same character string

(*1 blank*) 1975 (*10 blanks*) HELLO (*10 blanks*) -127.5

to be printed as the next line on the line printer.

READING FROM AN ASCII FILE

Reading from an ASCII file can be done with either a file READ statement or a file LINPUT statement. The file READ statement reads data as if from an INPUT statement on your terminal. For example, the following program reads a character string representation of number, a string, and a number from a card:

```
10 FILES ASC1
20 READ #1; X, A$, Y
30 END
```

The data on the card must be separated by commas and the string to be read into A$ must be enclosed in quotation marks, just as if all three items were being read in from an INPUT statement:

```
1975, "HELLO", -127.5
```

Frequently, data is not organized this way. The file LINPUT statement permits a program to read an entire record of an ASCII file into a string variable, without regard to internal commas or quotation marks.

```
20 LINPUT #1; A$
```

This statement would read an entire string, including blanks, commas, and quotation marks, into the one variable A$.

Note that simply printing data to an ASCII file will not necessarily enable that data to be read back from the file. The statement

```
20 PRINT #1; 1975, "HELLO"
```

will print the string

(1 blank) 1975 *(10 blanks)* HELLO

on file #1. A file READ statement to that same file

```
20 READ #1; X, A$
```

would return an error message because the two items in file #1 are not separated by a comma and the character string is not delimited by quotation marks.

EXAMPLE 9. The following program is an example of how ASCII files can be used to read a deck of cards from a card reader and print their contents on a line printer. First create two ASCII files equating the file ASC1 with a card reader and the file ASC3 with a line printer.

```
FILE-ASC1,CR0
FILE-ASC3,LP0

10 FILES ASC1,ASC3
20 DIM A$(80)
30 IF END #1 THEN 70
40 LINPUT #1;A$
50 PRINT #2;A$
60 GOTO 40
70 END
```

Statement Number	Meaning
10	Open the files ASC1 and ASC3.
20	Assign dimension 80 to A$ because A$ will be read from 80 column cards.
30	Set up an IF END condition to stop the program when the last card (an EOF card) is read.
40	Read an entire record from file #1 (e.g. read a card) into the string A$.
50	Print the string A$ on file #2 (the line printer).

The CONTROL function (CTL) permits additional user control of ASCII devices. It is used in ASCII file PRINT statements to cause the line printer to skip lines or pages or to suppress paper advances; it can advance or rewind magnetic tapes; it can be used to reset the pointer in an ASCII disc file; and it can control parity and record separators when punching a paper tape. (Refer to the CTL function and ASCII File Print in Section XI for a complete description of the CTL capabilities.) For example, the function CTL(24) sent to a magnetic tape unit will rewind the tape currently mounted on the unit. If CTL(24) is sent to an ASCII disc file, it resets the file pointer to the beginning of the file. CTL(24) is not defined for any other non-sharable device and will be ignored if it is sent to one.

EXAMPLE 10. The following program reads a deck of cards printing 25 cards on each line printer page. The CONTROL function CTL(1) advances the paper to the top of the next page.

```
10  FILES ASC1,ASC3
20  DIM A$[80]
30  IF  END #1 THEN 100
40  FOR R=1 TO 25
50  LINPUT #1;A$
60  PRINT #2;A$
70  NEXT R
80  PRINT #2;CTL(1)
90  GOTO 40
100  END
```

ASCII file devices can also be accessed using the *OUT= file name* forms of the RUN, EXECUTE, LIST, and PUNCH commands (refer to Section XI). For instance, to obtain a listing of employee salaries on the line printer, run the program in example 8 with the following command:

```
RUN*OUT= ASC3*
```

where ASC3 has been equated to LP0. The file must be an ASCII file supporting output (e.g. line printer, paper tape punch, magnetic tape, disc), defined with prior use of the FILE command.

The following table summarizes the major differences between BASIC formatted files and ASCII files:

BASIC FORMATTED FILES	ASCII FILES
Always present on-line	Not always present on-line
Data can only be created or retrieved on-line through programs	Data can also be created and retrieved off-line fairly easily
Numbers and strings can be intermixed without ambiguity	Numbers and strings may not be distinguishable outside of the context of an accessing program
Files can be accessed serially or directly	With few exceptions, files can only be accessed serially
Individual records can be altered without affecting other records	The ability to selectively alter portions of a file is restricted or non-existent, depending on the device being used
Always on disc	Can be on disc, magnetic tape, line printer, card reader, paper tape punch, or paper tape reader
Read and write access	Except for ASCII disc and magnetic tape files, a given file is read-only or write-only
Multiple access possible	No multiple access possible (hence the term "non-sharable device")

FORMATTED OUTPUT

SECTION VI

WHAT IS FORMATTED OUTPUT?

Formatted output gives you explicit and exact control over the format of your program output.

- Numbers can be printed in three different representations: integer, fixed point, and floating point.
- The exact position of plus and minus signs can be specified.
- String values can be printed in specified fields and literal strings and blanks can be inserted wherever needed.
- You can have full control over carriage returns and line feeds.
- Arbitrarily long lines can be printed without the carriage returns and line feeds that PRINT and MAT PRINT statements normally provide.

HOW DO YOU INDICATE FORMATTED OUTPUT?

There are two key elements in formatted output, the list of items to be printed (the using list) and the format in which these items are to be printed (the format string).

Example:

```
10 PRINT USING format string; using list
```

USING LIST

The using list specifies what items are to be printed. A using list is made up of the same kinds of items that can be included in the PRINT and MAT PRINT statements. The items are expressions, numeric constants, string variables, and print functions. Entire arrays can be formatted using the MAT PRINT USING statement.

Examples of using lists:

```
10 PRINT USING 100;        A,SIN(X),F$
20 MAT PRINT USING 200;    G
30 MAT PRINT USING 240;    Z,B
40 PRINT USING 150;        (L3+N)/R(I),SPA(N4),A4
                           ^^^^^^^^^^^^^^^^^^^^^
                           Print using list
```

FORMAT STRING

As the using list tells which items are to be printed, the format string specifies how the items are to be printed. A format string can be made up of a carriage control character to control carriage returns and line feeds; format specifications to indicate the format for each item in the using list; and delimiters to separate the carriage control character from the format specifications and the format specifications from each other.

A format string can be in a literal string included in the PRINT or MAT PRINT USING statement, in a string variable, or in a special statement called the IMAGE statement.

Examples of format strings:

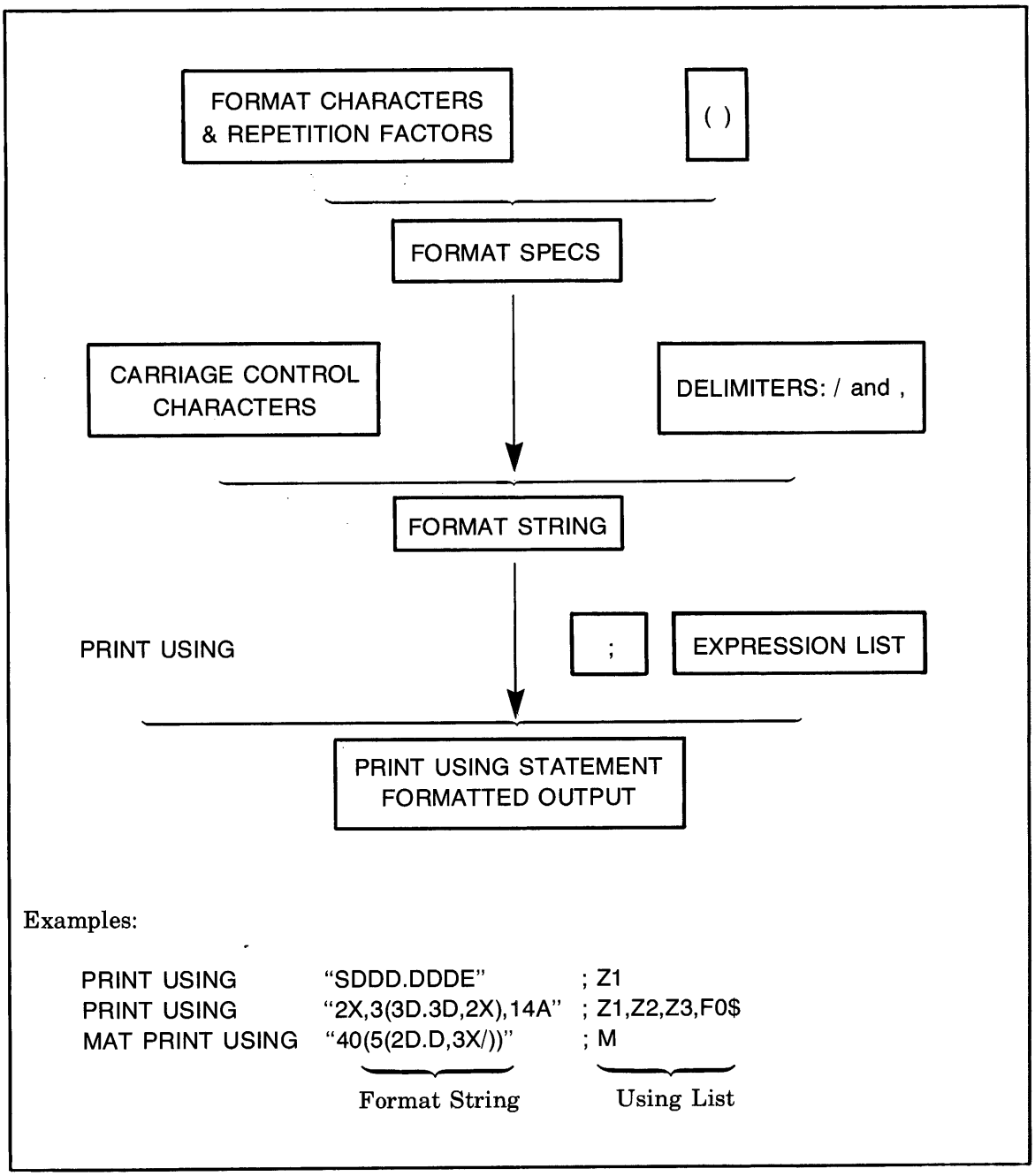
```
10 PRINT USING       "-,DD.D,3X,2(20A)" ; P4,A$,B$
20 MAT PRINT USING   "4(/5(SDDDD))"     ; T
30 PRINT USING       R$                 ; A(I),F2,TAB(3)
40 MAT PRINT USING   F1$(30)            ; G,M,X
50 IMAGE             #,2X,3(4E,2A),"---"
60 IMAGE             2X,6A/2X,6A//
                     ~~~~~~~~~~~~~~~~~~
                        format string
```

A statement number referencing an IMAGE statement may be used instead of the format string.

Example:

```
                  statement
                  reference      using list
                  ~~~~~~~~~      ~~~~~~~~~~
10 PRINT USING 20;       A, SIN(X), F$
20 IMAGE #, 3DX/, D.DDD, 3AX
```

Figure 6-1 illustrates how the various format and control characters are used to make up a PRINT USING statement.



Examples:

```
PRINT USING       "SDDD.DDDE"          ; Z1
PRINT USING       "2X,3(3D.3D,2X),14A" ; Z1,Z2,Z3,F0$
MAT PRINT USING   "40(5(2D.D,3X/))"    ; M
                   Format String         Using List
```

Figure 6-1. PRINT USING Statement Structure

USING FORMATTED OUTPUT

The first step in producing formatted output is to decide what each item in the using list should look like when it is printed. There are two basic kinds of data that can be formatted, numbers and strings.

NUMBER REPRESENTATION

A number can be printed as an integer (no decimal point), a fixed-point number (a decimal point in a fixed position), or a floating point number (either an integer or a fixed-point number followed by an exponent). The format characters that are used to specify numeric formatting are "D", "E", "S" and ".". Each occurrence of a "D" says that one digit should be printed. Including an "E" says that an exponent should follow the number. An "S" character says to print either a plus or minus sign (depending on the sign of the number). A "." specifies that a decimal point should be printed and also indicates where it should be printed. In addition, an "X" may be used to denote blanks in a format specification. A "D" or "X" may be preceded by a number to indicate repetition.

Integer format examples:

```
DDDD
4D
2DDD      } equivalent
2D2D

2DX3D
SDDD
S4D
DX3DS
```

Integer output examples:

Format Specification	Value	Format of Output
4D	1234	1234
S4D	1234	+1234
4DS	1234	1234+
5D	1234	1234
4D	1234.8	1235
DXDDD	1234	1 234
S1ØD	1234	+1234
DSDDD	1234	1+234
5D	-1234	-1234
4D	1234.2	1234

Fixed-point format examples:

```
DDD.DDD
DDD.3D
3D.3D
3D.DDD
```
} equivalent

```
S3D.3D
DXDXDX.DDXD
XD6X4D.8D
DDSDD.3D
```

Fixed-point output examples:

Format Specification	Value	Format of Output
3D.4D	465.465	465.4650
4D.2D	465.465	465.47
4D.3D	-465.465	-465.465
SDD2D.D	465.465	+465.5
S2D.4D	.465	+0.4650
S.4D	.465	+.4650
D.4D	-.465	-.4650
2D.4D	-.465	-0.4650

Floating point format examples:

```
SD.5DE
DDD.DDDXEX
SD.8DXE
S6DE
S6D.E
S6D.XE
S6D.DDDE
```

Floating point output examples:

Format Specification	Value	Format of Output
SDXE	4.82716×10^{21}	+5 E+21
DDDD.DDE	SAME	4827.16E+18
S5DX.X5DEX	SAME	+48 . 27159E +20
SD.5DE	SAME	+4.82716E+21
S.10DE3X	SAME	+.4827159382E +22

CARRIAGE CONTROL

There are three characters which can be used before the first format specification to indicate carriage control.

- A "+" character will suppress the line feed which would normally follow a print.
- A "−" character will suppress the normal carriage return.
- A "#" character will suppress both carriage return and line feed.

Omission of the carriage control character specifies that an X-OFF, carriage return, line feed sequence will follow the PRINT USING or MAT PRINT USING statement.

Examples:

```
10 PRINT USING "#,DDD,2X,AA";A,A$
20 IMAGE −,2A,3X,4D
```

LITERAL STRING

A literal string (any combination of characters not including a quote mark and enclosed by quote marks) can be included as a format specification and is printed as it appears.

Example:

```
400 IMAGE "TOTAL = ", X, S3D.2D
```

DELIMITERS

A delimiter serves to set off the format specifications and can be either a comma or a slash. A comma serves no purpose other than to delimit format specifications. A slash can delimit format specifications and, in addition, generates an X-OFF, carriage return, line feed sequence.

Example:

```
500 IMAGE − , 3A / 3D , 3D
```

PRINT FUNCTIONS

Print functions can be used in PRINT USING and MAT PRINT USING statements (and can also be used in all other PRINT statements). There are four print functions:

TAB(X) — Tabs out to column X before printing next item.

SPA(X) — Skips X spaces before printing next item.

LIN(X) — Skips X lines before printing next item. If X is negative, no carriage return is generated; if X is zero, only a carriage return is generated.

CTL(X) — Sends the argument X to the device specified in the print statement.

The CTL function is only used for ASCII files (an ASCII file can be used to direct output to a peripheral device). Each ASCII file type interprets its own control codes. Refer to Section XI for detailed information on the CTL function.

STRING REPRESENTATION

A string can only be printed as a sequence of characters. The format character which is used to specify string output is an "A". In addition, the format character "X" can be used to denote blanks in a format specification. An "X" or "A" format character may be preceded by an unsigned integer to indicate repetition.

String format examples:

```
AAAA
4A          } equivalent
2A2A

4X          special case (all blanks, so no variable required)
AXAXAXA     alternate characters and blanks
2X20A
```

String output examples:

Format Specification	Contents of String Variable	Format of Output
6A	ABCDEF	ABCDEF
5A	ABCDEF	ABCDE
8A	ABCDEF	ABCDEF
2X6A	ABCDEF	ABCDEF
AXAXAXAXAXA	ABCDEF	A B C D E F

The following example program prints out a numeric conversion chart. It prints a value in inches (from 1 to 25) and then prints the equivalent value in metres, centimetres, and micrometres. Integer, fixed-point, and floating point formats are used.

Program example:

```
100 REM PRINT THE HEADING
110 PRINT USING 120
120 IMAGE "INCHES",10X,"METRES",4X,"CENTIMETRES",4X,"MICROMETRES",//
130 FOR I=1 TO 25
140 REM PRINT THE VALUE OF INCHES
150 REM USING CARRIAGE CONTROL TO SUPPRESS CARRIAGE RETURN AND LINE FEED
160 PRINT USING 170;I
170 IMAGE #,2X,2D,13X
180 REM CONVERT TO OTHER UNITS
190 M=I*.0254
200 C=M*100
210 M1=M*1.E+06
220 REM PRINT VALUES IN FIXED POINT AND FLOATING POINT
230 PRINT USING ".DDD,8X,DD.2D,8X,D.3DE";M,C,M1
240 NEXT I
250 END
```

```
RUN

INCHES          METRES     CENTIMETRES     MICROMETRES

   1              .025         2.54         2.540E+04
   2              .051         5.08         5.080E+04
   3              .076         7.62         7.620E+04
   4              .102        10.16         1.016E+05
   5              .127        12.70         1.270E+05
   6              .152        15.24         1.524E+05
   7              .178        17.78         1.778E+05
   8              .203        20.32         2.032E+05
   9              .229        22.86         2.286E+05
  10              .254        25.40         2.540E+05
  11              .279        27.94         2.794E+05
  12              .305        30.48         3.048E+05
  13              .330        33.02         3.302E+05
  14              .356        35.56         3.556E+05
  15              .381        38.10         3.810E+05
  16              .406        40.64         4.064E+05
  17              .432        43.18         4.318E+05
  18              .457        45.72         4.572E+05
  19              .483        48.26         4.826E+05
  20              .508        50.80         5.080E+05
  21              .533        53.34         5.334E+05
  22              .559        55.88         5.588E+05
  23              .584        58.42         5.842E+05
  24              .610        60.96         6.096E+05
  25              .635        63.50         6.350E+05

DONE
```

REPORT GENERATION

This program is a sample report generator. It first requests a school number from the terminal, then reads and prints out information about the school's teachers from a file. Note that a carriage control character is used to advantage (statement 100), slashes (/) are used (statement 200), string and fixed-point fields are used (statement 210), and an error occurs in the output for the eighth teacher (number too large for field; therefore, it is printed in E format on a separate line).

Program:

```
10 REM THIS PROGRAM GENERATES A REPORT ON TEACHERS
50 DIM A$(25),B$(19),C$(19)
60 FILES SCH1,SCH2,SCH3,SCH4,SCH5
100 IMAGE #,"ENTER SCHOOL NUMBER:"
150 IMAGE "TEACHER",13X,"SUBJECT",13X,"SALARY",4X,"ATTND."
175 IMAGE "-------",13X,"-------",13X,"------",4X,"------"/
200 IMAGE "CENTRAL CITY SCHOOL DISTRICT"/"DAILY REPORT OF ",25A//
210 IMAGE 20A,20A,"$",DDD.DD,DD.DDDD
230 PRINT USING 100
250 INPUT Z
260 READ #Z;A$,N
270 PRINT LIN(6)
500 PRINT USING 200;A$
550 PRINT USING 150
555 PRINT USING 175
557 FOR A1 =1 TO N
560 READ #1;B$,C$,A,B
600 PRINT USING 210;B$,C$,A,TAB(50),B
620 NEXT A1
1000 END
```

```
ENTER SCHOOL NUMBER:?1


CENTRAL CITY SCHOOL DISTRICT
DAILY REPORT OF B. BAKER HIGH SCHOOL


TEACHER              SUBJECT              SALARY       ATTND.

MISS BROOKS          ENGLISH              $450.34      12.5000
MISS CRABTREE        REM. READING         $400.00      64.3200
MISS GRUNDIE         HISTORY              $350.00       1.0010
MRS. HUMPREY         SPELLING             $700.00      99.9900
COLONEL MUSTARD      CRIMINOLOGY          $700.00      21.4500
MISS PEACH           LIFE PREPARATION     $232.00      23.2320
PROF. PLUM           AGRICULTURE          $777.77      65.0050
MISS H. PRYNNE       SOCIAL STUDIES       $100.25
+5.00500E+02

MISS SCARLETT        P.E.                 $205.10      25.0000
MR. SIR              HOME ROOM            $890.00      99.9000
MR. T. TIM           MUSIC                $ 10.99       0.0500
MR. WEATHERBY        ECONOMICS            $767.99      10.0400
```

SYSTEM FACILITIES

SECTION VII

This section contains an introduction to statements which can be used to control system operation from an executing program. Program execution can be controlled by the CHAIN statement and parameter passing by the COM statement. In addition, selected commands can be used from within a program by using the SYSTEM statement (refer to Section X for a detailed discussion of commands).

LINKING PROGRAMS

The Access system allows you to link programs so that one program can "call" another program for execution. This allows you to segment one large program into several smaller, easier to manage programs.

The CHAIN statement when executed causes the current program to terminate and a referenced program to be brought into your work area and start execution.

Example:

```
100 CHAIN "PROG2"
```

The CHAIN statement allows you to specify a return variable. If for any reason the destination program cannot be executed, the return variable is set to a value indicating the results of the chaining attempt.

Example:

```
100 CHAIN R, "PROG2"
110 REM CHAIN FAILED
120 GOTO R+1 OF 130, 150, 170
130 PRINT "BAD LINE NUMBER SPECIFIED"
140 STOP
150 PRINT "NO ACCESS PERMITTED"
160 STOP
170 PRINT "CHAIN NOT PERMITTED"
180 STOP
```

Normally when a chain operation is performed the destination program begins execution at its first statement. If you want to start at some other point in the destination program, you can specify the program entry point following the program name parameter.

Example:

```
100 CHAIN R, A$, 150
```

PASSING PARAMETERS

If you chain from one program to another you may want to transfer data values input or calculated in the first program. This can be done by specifying the string or numeric variables as being common to more than one program using the COM statement.

Example:

```
100 COM A$ (255), B(10,10), N, A0$(5)
```

The COM statement sets the dimension of the listed variables (performs the same function as a DIM statement) and in addition causes the system to retain their values during a chain operation. The destination program must also contain a COM statement. The variables listed in the COM statement are then set to the values passed from the previous program. The variable names in the destination program need not be the same as those used in the prior program. Values are assigned in the same order as they are named in the COM statements. For this reason you must insure that variables are of the correct data type (numeric, string, array), and of the same dimensions.

Example:

```
PROG1
10 COM A$(10),B,C(10,6)

PROG2
10 COM Z0$(10),D,R(6,10)
```

In this case the value of A$ is assigned to Z0$, the value of B is assigned to D, and 60 values for C are assigned (in a row, column order) to the 60 values of R.

EXECUTING PROGRAM COMMANDS

It is possible to execute some commands from an executing program using the SYSTEM statement. Commands which can be used with the SYSTEM statement are:

GROUP 1

BYE	(logs you off the system)
ECHO	(adjusts data transmission)
MESSAGE	(sends a message to the operator)
FILE	(creates an ASCII file)
PROTECT	(sets a file to the PROTECTED state)
LOCK	(sets a file to the LOCKED state)
PRIVATE	(sets a file to the PRIVATE state)
UNRESTRICT	(sets a file to the UNRESTRICTED state)
MWA	(sets a file to the Multiple Write Access state)
SWA	(sets a file to the Single Write Access state)

GROUP 2

TIME	(returns the *idcode, port number,* amount of current terminal, and total time used and time permitted for your *idcode*)
CATALOG	(returns a line of entries in your library)
GROUP	(returns a line of entries in your group's library)
LIBRARY	(returns a line of accessible entries in the system library)
LENGTH	(returns the length of the program in your work space, how much space you are using in your library, and the total space permitted in your library)

When the SYSTEM statement is used with a command in the first group, it uses a command string and a numeric return variable as parameters. The command string is the command to be executed. This may be a string variable containing the command or the command itself. The form of the command is exactly the same as it would be if you had entered it from your terminal. The return variable is used to indicate whether or not the command executed properly. The return variable is set to "0" for successful execution or to "1" if the command could not be executed.

Examples:

```
10 A$="MESSAGE — HELLO OUT THERE"
20 SYSTEM R1,A$
30 IF R1=0 THEN 50
40 PRINT "MESSAGE WAS NOT SENT"
50 SYSTEM R2, "FILE — LPRTR, LP0"
60 IF R2=0 THEN 80
70 PRINT "FILE NOT CREATED"
80 STOP
```

When the SYSTEM statement is used with a command in the second group it uses a return or destination string instead of a numeric return variable. Each of the commands in the second group would normally produce one or more lines of output at your terminal. When used in the SYSTEM statement only the first line of output (less any heading) is returned in the destination string. (*OUT=file name* is not permitted in this context.)

Example:

```
10 DIM A$(72)
20 SYSTEM A$,"LEN"
30 PRINT A$
40 END
RUN


00019 WORDS = 01 RECORDS.  01551 RECORDS USED OF 65000 PERMITTED.

DONE
```

When more than one line of output is required you can obtain successive lines of output by modifying the command. In the following example, the LIBRARY command is used to obtain a partial list of system programs and files. Each time a line of output is received, the starting name used in the command is modified to obtain the next sequential entry.

Example:

```
10  DIM A$[10],B$[72]
20  A$="LIB-"
30  INPUT A$[5,10]
40  IF A$[5,10] >= "TEST09" THEN 170
50  SYSTEM B$,A$
60  PRINT B$
65  A$[5,10]=B$[49,54]
70  FOR I=10 TO 5 STEP -1
80  IF A$[I,I]="9" THEN 130
90  IF A$[I,I]#"Z" THEN 150
100  A$[I,I]="0"
110  NEXT I
120  GOTO 40
130  A$[I,I]="0"
140  GOTO 40
150  A$[I,I]=CHR$(NUM(A$[I,I])+1)
160  GOTO 40
170  END
RUN


?PGM4
PP0     AL    PP0     64    PRIN     U      1          PTTTY   CU      2
PZ075    U      4          PZ125    U      6          PZ325    U     15
RATES   CU      2          SCOOP   CU      1          SCR     FL    100
SNLIST  FL    100          STATS   CU      8          T       FL     20
TESTER   U      1          TRADER   L     13          TRADES   L     29

DONE
```

SECURITY AND THE LIBRARY HIERARCHY

SECTION VIII

USER IDCODE ORGANIZATION

The system utilizes identification codes (idcodes) to identify every user. A unique password may be used along with each idcode to insure account security; but the system does not actually require its use. The idcodes are organized by the system for accounting purposes, controlling accessing capabilities, data security and account privacy. An idcode is made up of a single alphabetic character followed by three decimal digits. The range is from A000 to Z999 (inclusive); making a total of 26,000 distinct idcodes.

The system groups idcodes, 100 to each group. For example, A000 through A099 constitute the first group, A100 through A199 constitute the second group and so on through the last possible group, Z900 through Z999. There is a total of 260 groups.

User organization (by idcode) is illustrated in figure 8-1. The three concentric circles represent the three types of users as follows:

1. The very first idcode within the system, A000, is assigned to the user who will serve as the system master.
2. The first idcode within each group (for instance A100, D300, etc.) is assigned to the user who will serve as group master.
3. All other idcodes are available for assignment to private users.

Since A000 is both the first idcode in the system and the first idcode of Group #1, the system master serves as group master for the first group. (Duties of system and group masters will be discussed later.)

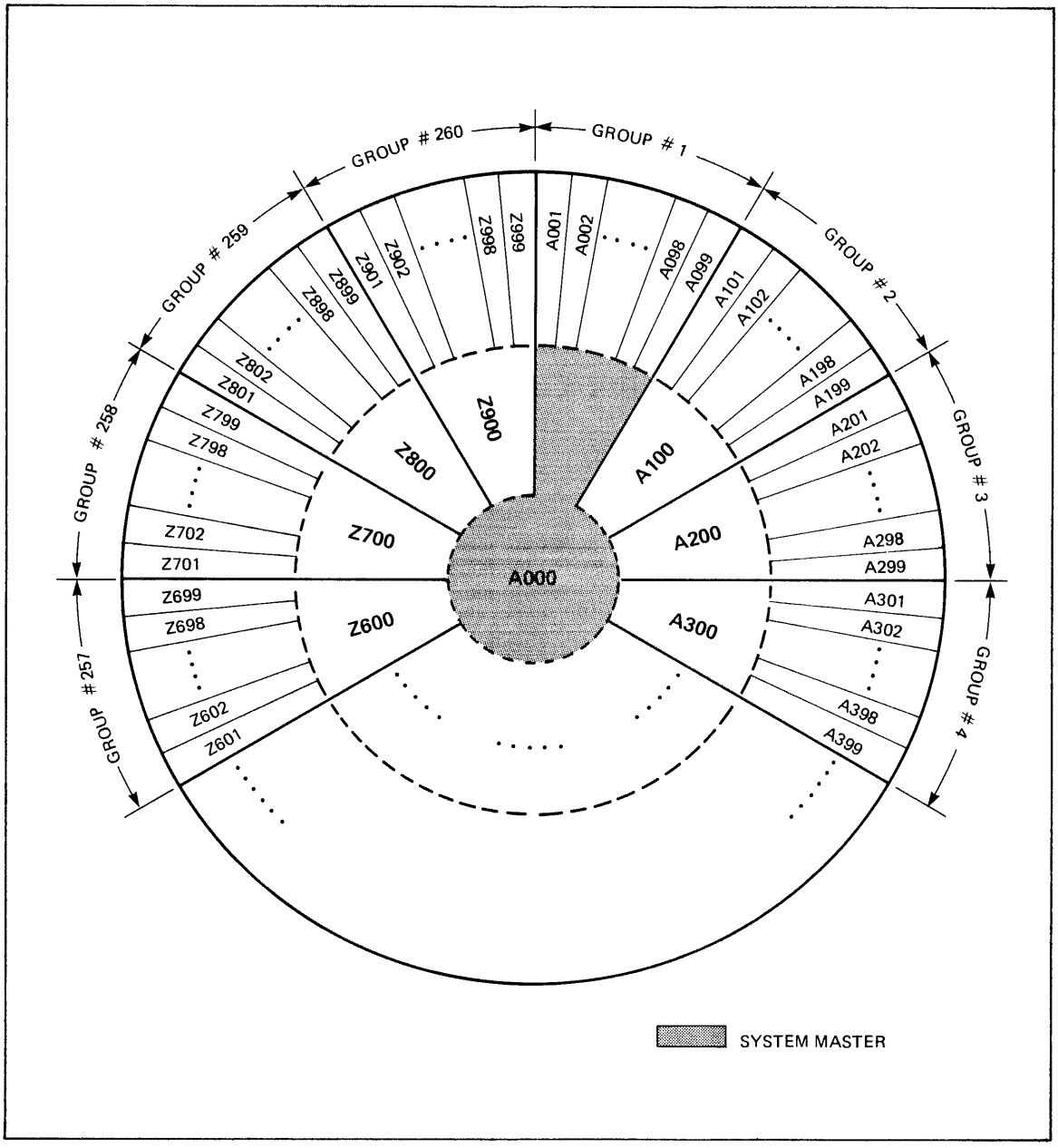


Figure 8-1. Idcode/Group Account Structure

When a user is assigned an idcode, an amount of disc space is allocated for storing programs and files. This disc space with its stored programs and files is called a library for that idcode.

Just as there are three types of users, so are there three types of libraries associated with those users.

1. The library assigned to the system master is referred to as the system library.
2. A library assigned to a group master is called a group library.
3. The library of every other user is known as a private library.

In figure 8-1, the outer ring of idcodes represents private libraries, the second ring of idcodes represents the group libraries, and the inner circle represents the system library. (Note that the system library is also the group library for Group #1.) Each user can access elements in his own private library and, to the limit of any restrictions, programs and files in his group library and in the system library. A user cannot access other private libraries or other group libraries unless they have the PFA capability (see Account Accessing capabilities).

PRIVATE LIBRARY — PRIVATE USER

A private library is created and maintained by each idcode. This library is completely controlled by the user assigned that idcode. The user can enter, modify, restrict access to, and delete programs and files within his private library. When the account has Program/File Access (PFA) capability, its library can (under certain circumstances) be accessed and altered by the group master or by other users.

GROUP LIBRARY — GROUP MASTER

A group library is a common library normally accessible only to members of the group. However, when the account has PFA, all or part of the library can be made accessible to all other system users. The group master is responsible for creating, maintaining, deleting, and controlling access to the programs and files within this library. The group master is also able to create, give MWA status to, read, write, and purge locked BASIC-formatted files in the accounts of group members having PFA — if he has been given the File Create/Purge (FCP) capability by the system operator.

SYSTEM LIBRARY — SYSTEM MASTER

The system library is a common library normally available to all users of the system. The system master may, however, place accessing restrictions on individual programs and files within this library.

Basic functions of the system master are similar to those of a group master, but more extensive. The system master can enter, modify, restrict access to, and delete programs and files in the system library. The system master is also the group master for accounts A001 through A099 and, as such, can be given File Create/Purge capability over their libraries.

The system master is responsible for creating and maintaining the optional HELLO program within the system library. This BASIC language program transmits pertinent information to users at logon time. The logon message might inform users of the following:

- System identification
- Port Number
- Date and time of logon
- System hibernate and sleep schedules
- News for the day
- News about new applications programs

It is possible to use the HELLO program to create a dedicated environment. If an account is to execute only one particular application program, and not be permitted any other system activity, the HELLO program can channel the user to the application program immediately upon logon. This might be accomplished in the following manner:

a. Establish a file that contains (for each account on the system) information about planned account usage. This file should reside in A000 and be locked; hence only available to locked system library programs.

b. Write the HELLO program so that it determines the user's logon idcode, scans the file previously established, and takes the user to the appropriate program. Or the HELLO program could CHAIN to some other program that performs this task.

c. The application program should disable the BREAK key at the user's terminal and execute a BYE command whenever it relinquishes control.

ACCOUNT ACCESSING CAPABILITIES

Thus far we have discussed the accessing capabilities the system automatically grants each idcode. However, there are other types of accessing capability not associated with the idcode/group structure. One is based on the account Program/File Access (PFA) capability as established by the operator via the NEWID or CHANGEID commands. When the system master assigns an idcode to a user, he may give that account the Program/File Access capability. Thus the account is considered to have PFA or NOPFA. When an account has PFA, the owner may make elements in his library accessible to all other users; forming a pseudo system library. In this way, a user may access private libraries in addition to his own group library and the system library.

An idcode may also be granted Multiple Write Access (MWA) capability via the NEWID or CHANGID operator commands. Account MWA capability makes it possible for an owner to declare his files to have MWA status. When no Multiple Write Access capability has been granted to an account by the operator, files in the account library always have SWA status. SWA stands for Single Write Access. When a file has SWA, only one user at a time can write on the file. When a file has MWA, several users can access and write to the file simultaneously.

When the system master creates a group master idcode, an option exists for granting the account File Create/Purge (FCP) capability. A group master's account is said to have FCP or NOFCP. With FCP, the group master has special accessing powers within the libraries of his own group members. He may create, give MWA status to, read, write, or purge locked BASIC-formatted files located in libraries of his own group members, so long as those libraries have the PFA (Program File/Access capability). The FCP capability is only operative from an executing program which has been saved in the group master's library.

Of course an owner can always limit access to his library by placing restrictions on individual programs and files.

USER IMPOSED RESTRICTIONS FOR PROGRAMS

There are four possible states for any program saved in a library: unrestricted, protected, locked, or private. The following only applies to program access by other users. You are always allowed to access any of your own programs. When a program is saved, its state is private. This means that no other user may access that program. Maintaining a program private guarantees its absolute privacy from other users. You are allowed to change the state of a program at any time.

A locked program may only be executed or chained to (no line number allowed) by other users. Other users are not allowed to have a local copy of any of your locked programs. Locked programs are useful in cases where you wish to allow other users to run your program, but not be able to list or modify it. Locked programs are also useful for file accessing applications. For example, your locked program may read and write on your locked files, regardless of who is executing the program. (Your locked files are otherwise unavailable to other users.)

Other users may obtain a local copy of your protected programs; however, they may not list the program. Other users are allowed to modify their local copy of your protected program. This is useful in cases where you supply the program and another user supplies his own data in the form of DATA statements. Another application for protected programs occurs when you have subroutines for general use in another user's main program. The other user may append your subroutines to his main program, but may not list them.

An unrestricted program in a library with the PFA capability can be accessed by any other user. Other users can GET, LIST, SAVE, and modify your unrestricted programs.

USER IMPOSED RESTRICTIONS FOR FILES

There are four possible states for any file saved in a library: unrestricted, protected, locked, or private. You are always allowed to access (both read and write) any of your own files. The following applies to file access by other users. When a BASIC formatted or ASCII file is created its state is locked. Only the owner or the owner's locked programs may access locked files.

> Note: Non-sharable devices may not be "loaned" except that a locked program saved in A000 may access locked non-sharable devices belonging to A000.

Making a file private guarantees its absolute privacy. No other user is allowed access. Group and system library listings do not even show private files (or programs) since no other user may access them.

Other users may only read a protected file. Protecting a file allows you to set up a data base which can be read by other users, but only updated by you.

Unrestricted files have no security. They may be read or written to by any other user.

CONTROLLING SIMULTANEOUS WRITING ON FILES

A user with the MWA (Multiple Write Access) capability can specify that a file can be written to by more than one user concurrently. Obviously, some means of controlling this access is necessary. There exist two statements to control this type of access: LOCK and UNLOCK.

Cooperating users may use the LOCK and UNLOCK statements to prevent writing on the same record at the same time, which might otherwise invalidate the write operation. The LOCK and UNLOCK statements do not actually "lock" or "unlock" a file. They set and clear "busy flags" associated with the named file. Execution of the LOCK and UNLOCK statements return the current status of this flag.

When you want to write on a MWA file which may already be in use, you should first LOCK the file. If another user already has the file LOCKED, your LOCK will not take effect until that user UNLOCKS the file. Cooperating users can then insure that they are not all trying to write on the file at the same time. (Do not confuse the statement LOCK with the locked state. The former is a facility available from a running program to prevent simultaneous writes; the latter is a state the file may be in which restricts any kind of access to the file.)

Considering idcode/group accessing capabilities together with the PFA, FCP, and MWA accessing capabilities and the protected, locked, unrestricted, and private states of programs and files; it is possible to plan elaborate data security and access schemes for your applications. Remember that the ninety-nine accounts numbered A001 through A099 are unique on an HP 2000 Access System since they have the system library as their group library. This feature might be utilized in a number of ways. For example, an applications program can be a locked program in the system library which in turn has the FCP capability. As such, it can programmatically create, manipulate, and purge files in the A0*xx* accounts, even when being run from some other account on the system.

REMOTE JOB ENTRY FACILITY

SECTION IX

As an Access user you can use the Remote Job Entry Facility to transmit jobs to another system. This section will give you an overview of the RJE facility and an explanation of how to use it.

WHAT IS REMOTE JOB ENTRY?

Remote Job Entry (RJE) is a system facility which allows you to submit a job from the Access system for processing on another system. In effect, you use the Access system as a remote terminal for the other system. The other system will be referred to as the host in the following discussion.

Programs contained in jobs may be written in any language (FORTRAN, COBOL, RPG, etc.) available on the host system. Programs can be assembled, compiled, or executed on the host system. The programs can be used to retrieve data files on the host system and transmit them back to the Access system. Data can be entered and formatted on the 2000 Access system and then sent to the host system for processing. The results of the processing can be returned to the Access system for post processing, storage, or output.

The RJE facility transfers data over the public telephone network or private leased lines at rates up to 4800 bits per second (approximately 480 characters per second). A functional diagram of the system elements used by the RJE facility is given in figure 9-1.

WHAT HOST SYSTEMS CAN YOU COMMUNICATE WITH?

The RJE facility can be used to communicate with selected IBM and CDC computer systems. The RJE facility functions by emulating a Multileaving Remote Job Entry workstation (IBM) or a User 200 Terminal (CDC). Table 9-1 contains a list of the host systems that can be accessed with RJE.

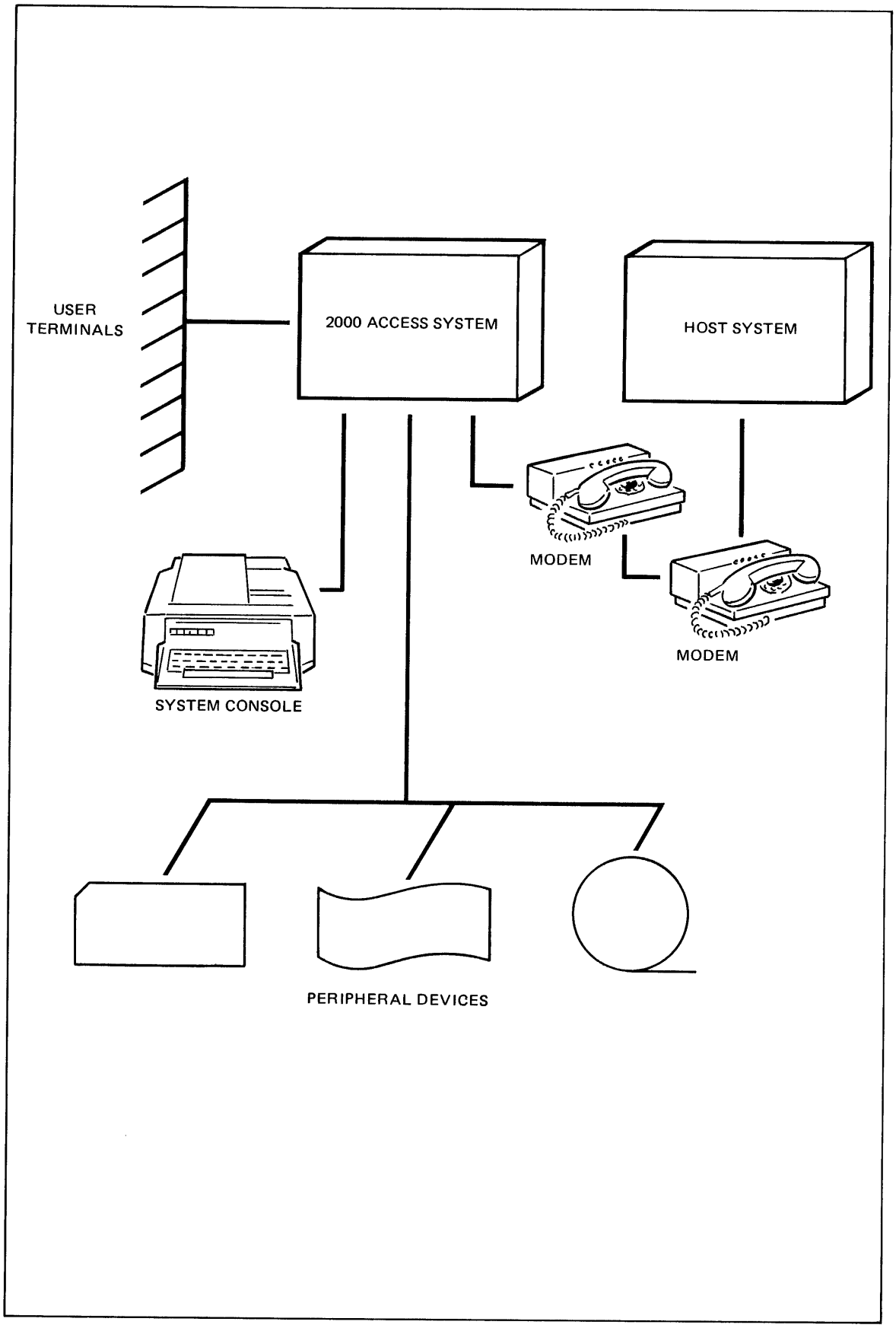


Figure 9-1. Elements of the 2000 Access Remote Job Entry Facility

Table 9-1. RJE Compatible Host Systems

IBM 360/370	CDC 3300/6400/6600/Cyber 70
OS/MFT/HASP	KRONOS/EXPORT/IMPORT
OS/MVT/HASP	SCOPE/INTERCOM
OS/MVT/ASP	
OS/VS1/JES/RES	
OS/VS2/JES2	
OS/VS2/JES3	

MULTILEAVING RJE WORKSTATION (MRJE/WS)

When the RJE facility is used with an IBM host system it emulates an MRJE workstation. An MRJE workstation is a remote batch input/output port operating under the multileaving protocol used with HASP. The remote port uses several data streams or functional lines for simultaneous input, output, and control communication. Table 9-2 contains a list of these functions. There are seven host reader functions for accepting input; seven host list functions for producing output; seven host punch functions for producing punched output; a host inquiry function for entering system requests or commands, and a host message function for receiving responses to system commands and system messages.

Table 9-2. IBM HASP Workstation Host Functions

HI1	Host Inquiry, used to send commands to the host system
HM1	Host Message, used to receive messages from the host system
HR1-7	Host Reader, used to enter jobs, normally a card reader
HL1-7	Host Lister, used to output jobs, normally a line printer
HP1-7	Host Punch, used to output jobs, normally a punch

USER 200 TERMINAL

When the RJE facility is used with one of the CDC host systems listed in Table 9-1, it emulates a User 200 Terminal. The User 200 Terminal is similar to the HASP workstation described previously but uses only one host reader, one host lister, one host inquiry, and one host message function (HR1, HL1, HI1, HM1).

HOW DOES RJE WORK?

When you use the RJE facility, portions of the Access system act as a remote port on the host system. The various host functions are assigned to various real or virtual devices. The exact number of host functions assigned by IBM host systems can be varied. Since only one host function of each type is used with CDC systems, that configuration is fixed. In either case the host inquiry and host message functions will always be used.

The RJE facility looks like a collection of input and output devices to the host system. A typical configuration for an IBM host is shown in Figure 9-2. Note that some of the host functions have been assigned to virtual devices or job function designators. A table of job function designators is given in Table 9-3. The job functions correspond to the host function types.

Table 9-3. Job Function Designators

JOB FUNCTION DESIGNATOR	DESCRIPTION
JI0	Job Inquiry Designator, allows you to send commands and messages to the Host Inquiry function in an executing program.
JM0	Job Message Designator, allows you to read messages from the Host Message function in an executing program.
JT0-6	Job Transmitter Designator, allows you to send jobs to a Host Reader function in an executing program.
JL0-6	Job Lister Designator, allows you to read the output of a job from a Host Lister function in an executing program.
JP0-6	Job Punch Designator, allows you to read the output of a job from a Host Punch function in an executing program.

These job function designators look like real devices to the host system but in fact merely pass input and output data to programs executing on the Access system.

When the Access system is configured, each of the host functions allocated to the remote port must be assigned to either a real device (card reader, line printer, paper tape punch) or to a job function designator (JT, JL, JP). The host inquiry and host message functions are always assigned to the Access system console. It is optionally possible to assign the host inquiry function to a job inquiry function and the host message function to a job message function. Other than the host inquiry and message functions, the assignments may be changed at any time by the system operator.

When you are ready to use the RJE facility, the system operator typically dials up the host system connecting the RJE modem. The operator then enters a signon command at the system console followed by any special configuration commands that may be required for a given job. Once this has been done the RJE facility is ready to accept input from card readers or from BASIC programs via the job transmitter function. Any output ready to be returned from the host system can be sent to printers, punches, or job list or punch functions.

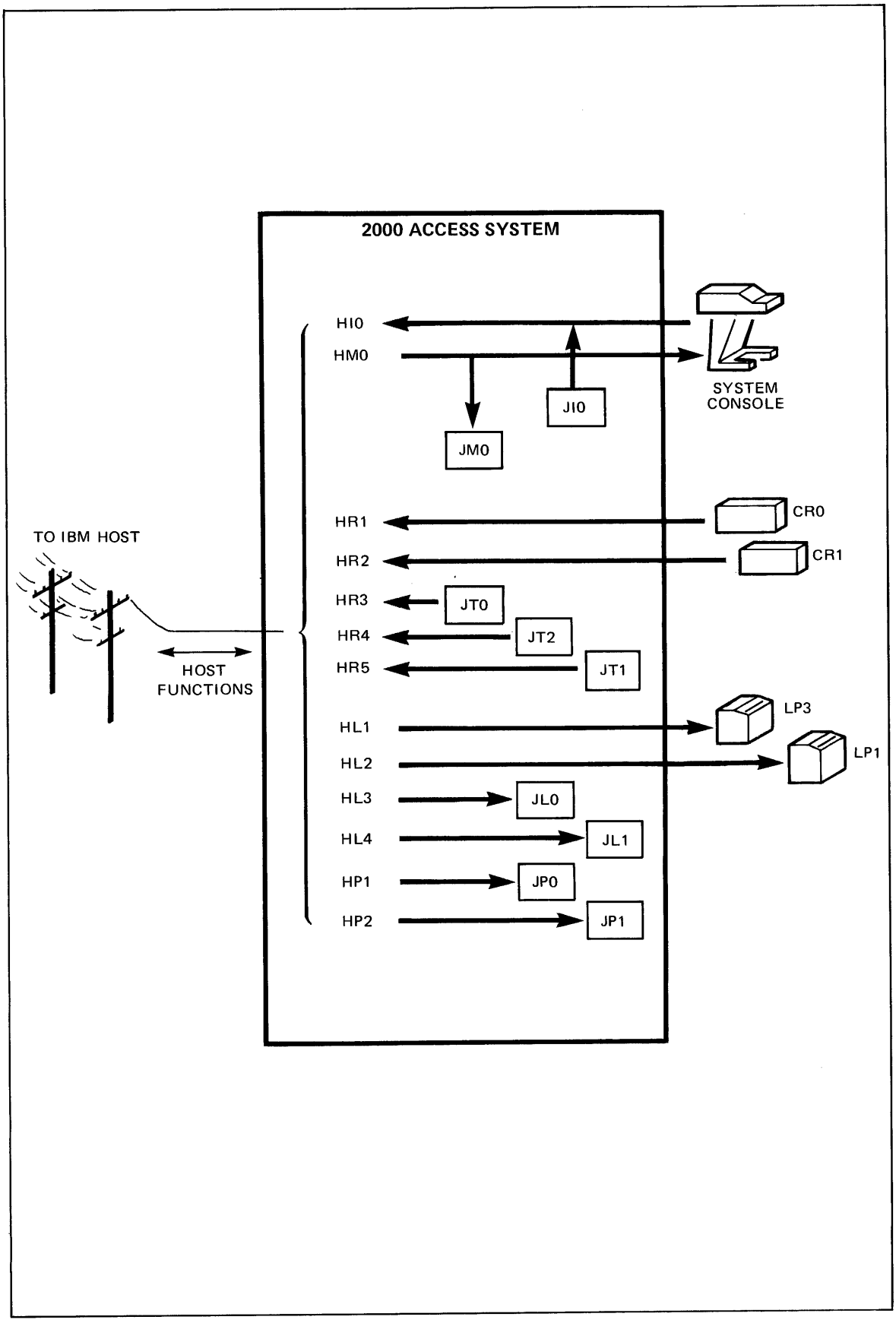


Figure 9-2. Typical RJE Configuration For An IBM Host System

Once the communications link has been established and the operator has signed on to the host system, the system console can be used to enter commands to be executed by the host system. These commands can be used to obtain job status, modify the configuration of the remote port, change job priority, or change the routing of job output. The commands available vary with different host systems.

You can also send host commands from an executing program by printing on the job inquiry file. You can receive any host system responses by reading the job message file. Techniques for doing this are discussed under HOW DO YOU USE REMOTE JOB ENTRY?

Tables 9-4 through 9-6 contain commands available to you for certain host systems. You should refer to the appropriate host system manuals for detailed explanations of the commands. A list of host system manuals is given in table 9-7.

Table 9-4. Summary of 360 HASP Remote Commands

COMMAND	COMMENTS
$DA	Display status of active jobs
$DF	Display information on jobs queued for printing with special forms
$DN	Display names and status of all jobs
$DQ	Display number of jobs in each queue type
$CJn or $C'name'	Cancel job with job number 'n' or job with job name 'name'
$DJn or $D'name'	Display status of job with job number 'n' or with job name 'name'
$PJn or $P'name'	Cancel specified job after completion of current activity
$Bdevice	Backspace device
$Cdevice	Cancel current function on device
$Edevice	Restart current function on device
$Fdevice	Forward space device
$Idevice	Interrupt current function on device
$Ndevice	Repeat current function on device
$Pdevice	Stop the device after completion of the current function
$Sdevice	Start the device
$Tdevice	Set device parameters (to indicate such things as type of forms loaded in a printer, or type of print character set in use)
$Zdevice	Halt the device immediately

Table 9-5. Summary of CDC EXPORT/IMPORT Remote Commands

COMMAND	COMMENTS
ON	Turns equipment logically on. Card readers are initially on, and output devices are initially off.
OFF	Turns equipment logically off.
DEFINE	Specifies the various attributes of an output device (print train, forms code, etc.).
WAIT	Temporarily halts reading or printing.
GO	Resumes operation after a WAIT command.
BSP	Backspaces a print file a specified number of sectors.
END	Terminates current operations on the specified equipment.
REP	Used to specify the number of additional copies of the file in process to be printed.
REW	Rewinds the print file in progress and turns the equipment logically off.
RTN	Returns a print file to the appropriate queue with its present priority or a newly specified priority.
SUP	Suppresses the spacing of a print file, so that the remainder of the file is single spaced.
DIVERT	Allows the user to divert output files of a remote job to the central site or another terminal.
DROP	Allows a user to drop a job which is currently in execution.
EVICT	Allows a user to eliminate a job from the input and/or output queues.
KILL	Allows a user to kill a job which is currently in execution.
PRIOR	Allows a user to change the priority of a file in the output queue.
REVERT	Cancels the effect of a DIVERT
H	Displays the contents of the terminal's input and output queues.

Table 9-6. Summary of Some IBM ASP Remote Commands

COMMAND	COMMENTS
X,name [, message]	Specifies to ASP system the name of the support program to be scheduled for execution.
C,name	Terminates currently active support function
I,A	Displays active jobs
I,B	Displays backlogged jobs
I,D	Displays disposition of I/O devices
I,J=job name	Displays information about specific jobs
I,J=job number	Displays information about specific jobs
Z,console name,test	Sends text message to another console

Table 9-7. Some Useful Host System Manuals

HOST SYSTEM	MANUAL	DESCRIPTION
IBM	HASP Operator's Guide GC27-6993	Description of HASP remote operator commands
IBM	ASP Operator's Guide GH20-1289	Description of ASP remote operator commands
CDC	CDC 200 User Terminal 82128000	Description of terminal operating procedures
CDC	Computer Systems Reference Manual 60100000	General system reference
CDC	INTERCOM Reference Manual 60307100	General system reference
CDC	SCOPE 3.4 Reference Manual 60307200	General system reference
CDC	KRONOS General Information Manual 60407100	General system reference

HOW DO YOU USE REMOTE JOB ENTRY?

The first step in using the RJE facility is to establish the communications link between the Access System and the IBM or CDC host system. This may be done by requesting the system operator to make connection or using the RJE facility during a time when RJE is scheduled at your site. When the communications link is established, the RJE facility is ready to use.

SENDING JOBS THROUGH THE CARD READER

The easiest method of sending a job to the host system is to send a card deck through a card reader. The card reader must be connected to the host function HR (Host Reader). In this case all that is required is to load the card deck in the card reader and start the reader. When the RJE facility is enabled the reader(s) assigned to RJE is automatically made ready. Your job will be automatically sent to the host system for processing. Figure 9-3 shows a typical RJE card deck for a FORTRAN compile and execute on an IBM system. When using the card reader you must use a card with "::" in the first two character positions as the last card in the deck to be read. This indicates the end of the job to the Access system.

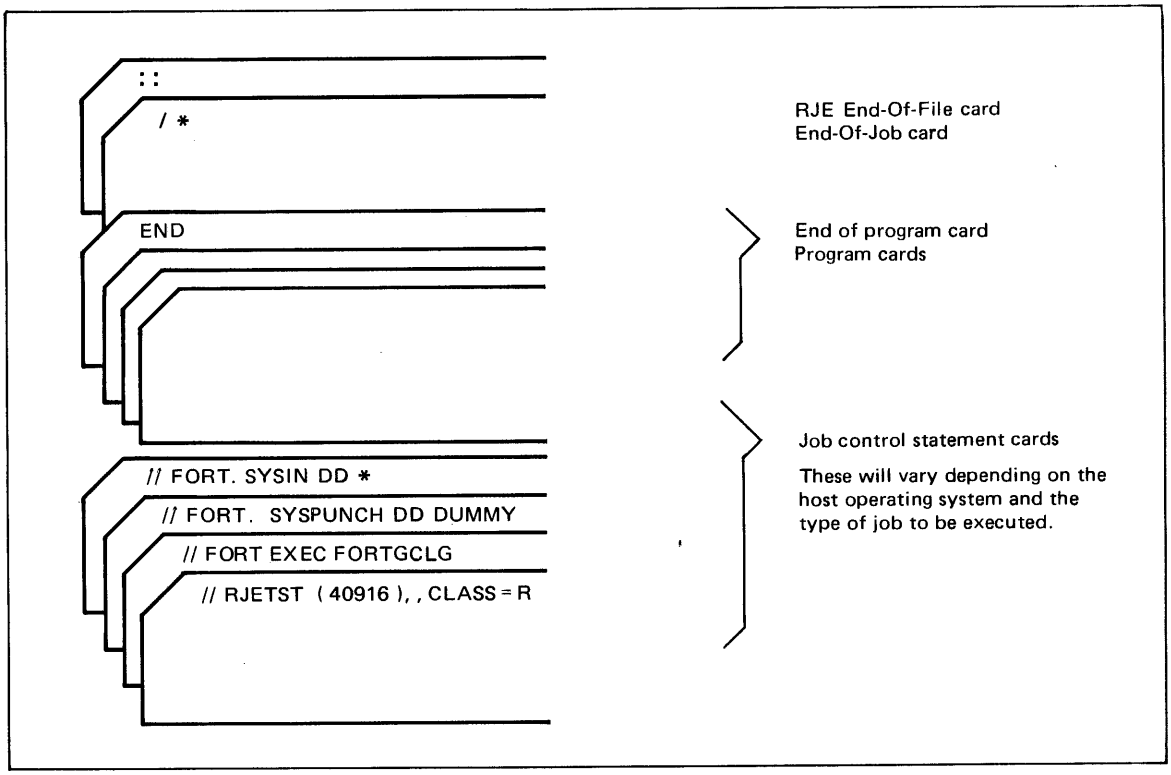


Figure 9-3. Example of an RJE Job Deck

RETRIEVING OUTPUT ON THE LINE PRINTER

When the host system finishes processing a job, the job's output is directed back to the Access system. If a line printer is connected to the host function HL (Host Lister), the job output is printed on the line printer. With CDC hosts there is no problem determining which host lister function the output will return on because CDC systems can have only one host lister function. IBM systems will normally return job list output to the first available host list function. Job punch output will be returned to the first available host punch function.

In order to ensure that output is returned to a specific host function you must use forms control commands. The IBM host systems allow you to designate specific host functions for certain output formats. This is done using the system operator's RJE command which sends a command on the host inquiry function.

The forms command is

$TRMx.PRn, F=f (for list output)

or

$TRMx.PUn, F=f (for punch output)

where

x = your remote station number (assigned by the host system operator)

n = the host function number that you want the forms output returned to

f = the forms number (four digits, x or # may be substituted as a "don't care" digit)

An example of forms assignment is given in Figure 9-4a. By using an 'x' or '#' in place of one or more digits of the forms number, a group of form types can be routed to the same host function with a single command.

Once forms assignment has been made, you can specify in your job deck (using job control statements) that output be printed using a specific form. The output will then be routed to the host function assigned to that form.

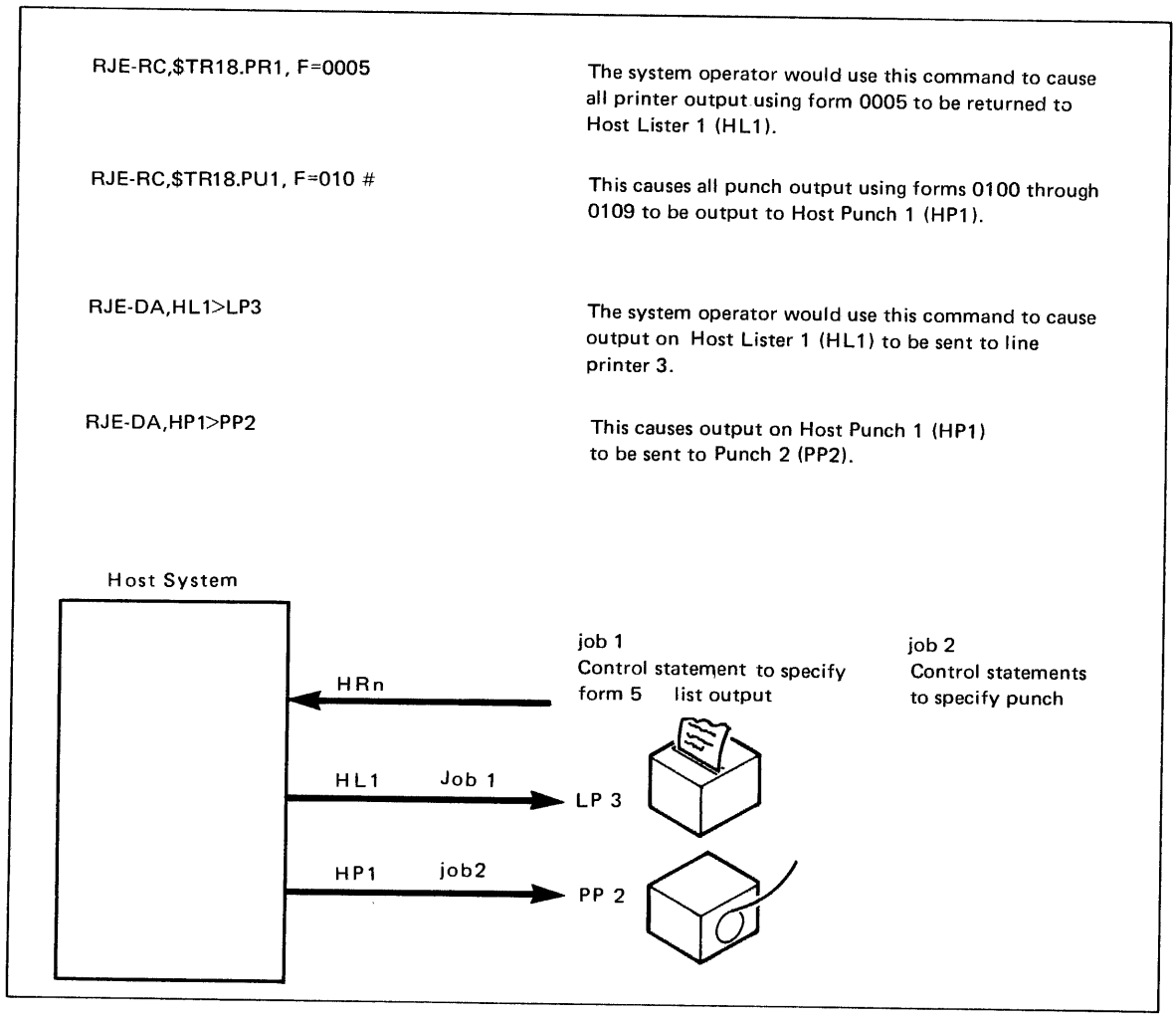


Figure 9-4. Example of Forms Assignment Used to Route Output

SENDING JOBS AND RETRIEVING OUTPUT FROM YOUR TERMINAL

Host reader, list, and punch functions (HR, HL, HP) need not be connected to actual devices. Instead, they may be connected to job function designators. In turn, a job function designator may be equated to a file name. A BASIC program manipulates these job functions just as if they were non-sharable devices. You may print to a job transmitter (JT), or read from a job lister (JL) or job punch (JP).

Sending a job from your terminal involves two steps. First you must determine what host functions are connected to which job function designators. Your system operator must provide this information as it differs from system to system. To make the following discussion easier, we will assume that you have access to all five types of job function designators and that they are connected as follows:

JI0 →HI1	Send messages and commands to host
JM0←HM1	Receive messages from host
JT0 →HR1	Send jobs to host reader one
JT1 →HR2	Send jobs to host reader two
JL0 ←HL1	Receive output from host lister one
JL1 ←HL2	Receive output from host lister two
JP0 ←HP1	Receive punched output from host punch one

After determining what host functions are connected to which job function designators, you are ready to transmit jobs from your terminal. As with any other non-sharable device you must first equate a file name with a job function designator.

Examples:

```
FILE-SENDER,JT0
FIL-TRANS1,JT1
```

You can now write a program to send a job. All that is necessary is to retrieve the lines that compose the job (from your terminal, data statements, other files, etc.) and to print them on a file equated to the job transmitter device.

Example:

The following program will send the job in the DATA statements to an IBM host system.

```
FILE-SEND,JT
LIST

10  FILES SEND                          Opens job transmitter file
20  DIM A$[80]
30  IF TYP(0) <> 3 THEN 50              If no more DATA statements,
40  STOP                                Stop
50  READ A$                             Read a line of the job
60  PRINT #1;A$                         Send it to the host
70  GOTO 20                             Loop
```

Actual Job to be sent

```
1000  DATA "// SENDJOB JOB (1234567,10,0), 'JEC'"
1010  DATA "// EXEC PGM=IEBGENER"
1020  DATA "//SYSIN DD DUMMY"
1030  DATA "//SYSPRINT DD SYSOUT=A"
1040  DATA "//SYSUT1 DD DSN=D0911274.ACCESS.SRCE(D.61),"
1050  DATA "    DISP = OLD"
1060  DATA "//SYSUT2 DD SYSOUT=A,DCB=(BLKSIZE=80,LRECL=80,RECFM=F)"
1070  DATA "/*"
9999  END
```

You may find out what job function designators you are allowed to use through the DEVICE command. If you attempt to run a program which accesses a job function designator without an RJE connection having been established, you will receive the message JT0-DEVICE NOT READY (substitute for JT0 the job function designator used). You may then abort your program with the BREAK key or ask the operator to establish RJE communications. Once established, your program will automatically continue (unless you abort it).

In the last example, only one job was sent. In order to send more than one job from a program, you must inform the system when there is no more to be sent for a given job. You may do this in two ways: close and reopen the job transmitter file; or execute a PRINT #n; END statement. Closing the file or printing an end-of-file tells the system that a job has been completed.

Example:

This program will send any number of jobs to a host system. The jobs are stored in ASCII files and the program asks the user what job should be sent next. The job name is also the name of the ASCII file where it is stored.

```
10 FILES SENDER,*                          Open job transmitter file
20 DIM A$(80)
30 PRINT "JOB NAME";                       Ask for job name
40 INPUT A$
45 IF A$="STOP" THEN 150                   If response is STOP, then stop
50 ASSIGN A$,2,R                           Open file where job is
60 IF R<3 THEN 90
70 PRINT "UNABLE TO ASSIGN FILE ";A$       If unable to open, send a message
80 GOTO 30
90 IF END #2 THEN 130                      When finished sending job, go to line 130
100 LINPUT #2;A$                           Get a line of the job
110 PRINT #1;A$                            Send it to the host system
120 GOTO 100                               Loop
130 PRINT #1;END                           Tell system, job is finished
140 GOTO 30                                Go ask for another job
150 END
```

Retrieving output from a job on your terminal is just as simple. Again you must determine what host functions are connected to which job function designators. After that you must equate an available job function designator (one you have the capability to use) with a file name.

Examples:

```
FILE-LIST,JL1
FIL-PUNCH,JP0
FILE-GETJOB,JL0
```

You can now write a program to retrieve job output. All that is necessary is to read the job list or job punch file and do what you want with the lines of output received.

Example:

The following program will retrieve job output and store the results in a BASIC formatted file.

```
FILE-GETJOB,JL0
CREATE-JOBOUT,400
LIST

10  FILES GETJOB,JOBOUT                    Opens job list file and output file
20  DIM A$[134]
30  IF  END #1 THEN 70                     When no more lines available, stop
40  LINPUT #1;A$                           Ask for a line
50  PRINT #2;A$                            Write it to output file
60  GOTO 40                                Loop
70  PRINT " JOB SUCCESSFULLY RETRIEVED"    Inform user of completion
80  END                                    Stop
```

When a host system returns lines of output to an RJE workstation, it includes information about spacing of lines. If you retrieve your job output on a line printer, the Access system automatically formats the lines. If you receive your job output on your terminal, you may wish to print the output on a line printer at some future time. In order to allow for this, the Access system includes the line printer control functions as part of each line it passes through the job list device. The first character of each line is a digit. If you convert this digit to a number and use it in a CTL function when printing to the line printer, your output will be formatted exactly as if the Access system were printing it. The second character of each line contains a null character. Actual print data begins with the third character of the string.

Example:

The following program will retrieve job output from a host system and print it on a line printer just as if the host system were printing it on a local line printer.

```
FILE-PRINTR,LP0
FILE-JOBLIS,JL
LIST

10  FILES JOBLIS,PRINTR                    Open job lister and line printer
20  DIM A$[134]
30  IF  END #1 THEN 70                     Stop when no more output
40  LINPUT #1;A$                           Get a line of output from host
50  PRINT #2;CTL(NUM(A$[1,1]));A$[3]       Tell line printer to space and print line
60  GOTO 40                                Loop
70  PRINT "JOB RETRIEVED"                  Tell user all done
80  END                                    Stop
```

IBM hosts expect a job punch device to select one of two options when punching output. This control information is not used when the Access system punches output on a local paper tape punch. However, the system does provide this control information when passing the output data to a job punch device. The mechanism is the same as with job list devices except that the first character will only contain a "1" or a "2". CDC hosts do not send this character since a User 200 Terminal does not include a punch function.

COMMUNICATING WITH A HOST SYSTEM FROM YOUR TERMINAL

Just as it is possible to send jobs and retrieve output using a program at your terminal, you may send messages and commands and receive messages from the host system with your BASIC program. Printing strings to the job inquiry file causes those strings to be sent to the host system as if they were typed on the Access system console. Any messages the host system sends to the Access system can be read from the job message file and printed at your terminal.

Example:

This program allows a user to send status commands to an IBM host system and to retrieve the returned status.

```
FILE-INQUIR,JI
FIL-RESPND,JM
LIST
```

```
10  FILES INQUIR,RESPND                    Open job inquiry, message file
20  DIM A$[120],B$[20]
30  PRINT LIN(1);"WHICH JOB'S STATUS DO YOU WANT";   Prompt user
                                                     for job name
40  INPUT B$                                Stop when user types STOP
45  IF B$="STOP" THEN 90
50  PRINT #1;"$D'";B$;"'"                   Send $D'name' command
60  LINPUT #2;A$                            Ask for response
70  PRINT A$                                Print it on user terminal
80  GOTO 30                                 Loop
90  END
RUN
```

```
WHICH JOB'S STATUS DO YOU WANT?DS2001
RJE $*15.11.20 JOB 617 DS2001       EXECUTING N PRIO 4

WHICH JOB'S STATUS DO YOU WANT?STOP
DONE
```

Refer to Tables 9-4, 9-5, and 9-6 for a summary of representative HASP, ASP, and EXPORT/IMPORT commands.

Whenever a line is sent to a job inquiry device, the line is also printed on the Access system console. This informs the system operator what a user is sending on the job inquiry function. Any messages sent by the host system are always printed on the system console.

Because any message sent to the system console is also sent to the user in control of the job message file, certain care must be exercised. In our example above, an IBM HASP Display Job command was sent and the response was then retrieved. If the operator had already sent a command which produces several lines of output, the user could have gotten one of these lines rather than the actual status requested.

HOW TO GET *YOUR* OUTPUT

Previous discussions indicated that retrieving output is simple. There is no problem if only one host list function is connected to one job list designator. However, for IBM systems only, it is possible to have several host list functions (up to seven) connected to any combination of real line printers or job list designators. The same applies to the multiple host punch functions. When the host is ready to return the output of a job, it generally selects the first available host list function for list output or the first available punch function for punch output. This means that if you send your job on job transmitter 3 (JT3), you are not guaranteed that you will receive the job's output on job lister 3 (JL3). You must use forms control to guarantee what host list function your output is returned to. You must also specify in your job that it must be printed (or punched) only on a device loaded with special forms.

Refer to the previous discussion of forms control for a general overview of how you can select a specific host list function for job output. You need not use the system console to issue forms control commands, however. The following example provides a program which can be used to set forms control.

```
FILE-INQ,JI                                  INQ = Job Inquiry
FILE-TRANS,JT3                               TRANS = Job Transmitter
FILE-OUTPUT,JL4                              OUTPUT = Job Lister
FILE-PAYROL,DS,100                           PAYROL = ASCII disc file
LIST


10  FILES INQ,TRANS,OUTPUT,PAYROL            Open files
20  DIM A$[134]
30  PRINT #1;"$TRI8.PR4, F=0005"             Set forms control on Host Printer 4
40  IF TYP(0)=3 THEN 80                      When out of data, go get output
50  READ A$                                  Send to the host one
60  PRINT #2;A$                                 line of the job
70  GOTO 40                                  Loop until job sent
80  ASSIGN *,2                               Close transmitter to signal end of job
85  IF  END #3 THEN 120                      When output done, tell user
90  LINPUT #3;A$                             Get a line of output and
100  PRINT #4;A$[3]                             write it to ASCII disc file
110  GOTO 90                                 Go get another line of output
120  PRINT "JOB SENT AND RETRIEVED"          Tell user that we are all done
130  STOP
500  DATA "// PAYROLL JOB (45678912),'PROGRAMMER'"
510  DATA "/*PASSWORD              SECRET"
520  DATA "// PAYUPDAT"                                          Actual
530  DATA "JOHN SMITH,40"                                        job to
540  DATA "MARY JONES,48"                                        be sent
550  DATA "HARRY DOAKES,32"
560  DATA "WILLIAM BLOOM,42"
570  DATA "/*"
9999  END
```

Note, in the above example, the procedure PAYUPDAT is assumed to contain the necessary job control statements to route output to a printer using form 0005.

COMMANDS

SECTION X

This section contains a definition of what 2000 Access user commands are and how they are used. General terms used to discuss the commands are defined early in the section. Following the term definitions a complete description of each user command is given. (Commands used by the System Operator are defined in the operator's manual.)

WHAT IS A COMMAND?

Commands instruct the system to perform control or utility functions such as storing and listing programs or logging on and off the system. Commands differ from the statements used to write a program in the BASIC language in that a command instructs the system to perform some action immediately, while a statement is an instruction to perform an action only when the program is executed or run. A statement is always preceded by a statement number; a command is not.

Any command can be entered once the logging on procedure has been successfully completed. Commands can be abbreviated to the first three characters. (Embedded blanks are ignored.) If a command is misspelled, the system will return three question marks indicating that it did not recognize the command. If a command is not received properly, the system will print "TRANSMISSION ERROR.REENTER". You should retype the command. Following entry of each command, (*return*) must be pressed to signal that command entry is complete.

Most commands have parameters to further define command operation. For instance, the LIST command causes a display of your current program. It may have parameters to specify that only part of the program is to be printed, or that the program is to be listed to a device other than your terminal. If parameters are used, a hyphen or the "*OUT= file name*" construct (refer to specific commands for a discussion of this form) is used to separate the command from its parameters. If multiple parameters are used, they are usually separated by commas.

Example:

```
LIS-20,100
```

This would list the program beginning with statement 20 (or the first statement with a number greater than 20) and continuing through statement number 100.

TERMS USED IN THIS SECTION

The following pages contain definitions of terms that will be used in the command descriptions given later in this section. These terms will be used to define in detail the operation of each of the commands.

ASCII File

An *ASCII file* is an area on the disc or a *non-sharable device* which can be accessed through file READ and PRINT statements. ASCII files are created with the FILE command. The FILE command equates a *file name* with either an area on the disc (ASCII disc file) or with a *non-sharable device*. ASCII files can contain only string data. Note that a number can be represented by a string (e.g., "123.45" represents the number 123.45).

Examples:

```
FILE-DFIL, DS, 40       (creates an ASCII disc file of 40 blocks)
FIL-PUNCH, PP           (creates an ASCII file equated to the paper tape punch)
FIL-READER, CR, 40      (creates an ASCII file equated to the card reader)
```

Use of ASCII disc files is granted to every user on the system (provided that their *idcode* has been granted the use of sufficient disc space). The system operator grants use of *non-sharable devices* to an *idcode* through the NEWID or CHANGEID commands.

You can access ASCII files as if they were your terminal. Each *record* of an ASCII file is read and/or written as if it were a line on a terminal.

BASIC Formatted File

A *BASIC formatted file* is an area on the disc which can be accessed through file READ and PRINT statements. BASIC formatted files are created through use of the CREATE command or the CREATE statement (refer to Section XI for a discussion of the CREATE statement). The CREATE command associates a *file name* with an area on the disc. BASIC formatted files can contain both string and numeric data. In addition, these files contain internal information about the type of each data item (string or number) in the file..

Examples:

```
CREATE-BFIL, 20         (creates a BASIC formatted file of 20 blocks)
CRE-BIGFIL, 2000        (creates a BASIC formatted file of 2000 blocks)
CRE-SREC, 100, 64       (creates a BASIC formatted file of 100 blocks of 64 words each)
```

Use of BASIC formatted files is granted to all users whose *idcode* has been granted disc space through the system operator's NEWID and/or CHANGEID commands.

BLOCK

A *block* is the physical unit of storage on the disc. All of the discs on a system are organized as a contiguous set of *blocks*. Each block contains 256 words (a number occupies two words; a character occupies one half of a word). When a *BASIC formatted file* is created, you specify how many *records* long it should be. Each record will be stored in one *block*. When an *ASCII disc file* is created, you specify how many *blocks* long it should be. One difference between an *ASCII disc file* and a *BASIC formatted file* is that the former can store more than one *record* per *block*.

DEVICE DESIGNATOR

A *device designator* is a two or three character mnemonic which specifies either a *non-sharable device* or a *job function designator*. There are two types of *device designators*, general and specific.

A *general device designator* specifies *non-sharable devices* and *job function designators* by class.

NON-SHARABLE DEVICE	GENERAL DEVICE DESIGNATOR
magnetic tape	MT
line printer	LP
card reader	CR
paper tape reader	PR
paper tape punch	PP

JOB FUNCTION DESIGNATOR	GENERAL DEVICE DESIGNATOR
job transmitter	JT
job lister	JL
job punch	JP
job inquiry	JI
job message	JM

A *specific device designator* names a specific *non-sharable device* or *job function designator*. *Specific device designators* are formed by appending a digit, 0 through 6, to the *general device designator*. A list of all *specific device designators* available to your *idcode* can be obtained with the DEVICE command.

Examples:

LP0, CR3, JM1, PP6, JT4, etc.

END-OF-FILE MARK (EOF)

An *end-of-file mark* is a logical indication stored in a file which indicates the current end of data. When created, each record of each file has an *end-of-file mark* written into it as the first item in the record.

FILE LENGTH

The *file length* indicates the size of a file. *File length* is always specified in terms of the number of *blocks* the file occupies on the disc. The length of any file must be less than 32,767 *blocks*. The *file length* is also limited by the amount of disc space allocated to an *idcode* and the amount of disc space remaining in the system.

FILE NAME

A *file name* is a symbol composed of 1 to 6 characters or digits. It is used to identify a specific file in a *library*. In some cases the name can be optionally qualified. A preceding dollar sign ($) indicates the file is to be found in the *system library*. A preceding asterisk (*) indicates the file is to be found in the *group library*. Following the name with a period (.) and *idcode* indicates the file is to be found in the *library* of that *idcode*.

Examples:

FILE12, 3A, MYFILE, $SFILE, *GFIL, HERS.A312

FULL DUPLEX

Full duplex is a term used to denote a method of character transmission from your terminal to the system. *Full duplex* means that when a character is typed on your terminal, it is sent to the system, but is not printed (or displayed) at your terminal. When the system receives the character, it immediately sends it back to your terminal causing it to be printed (or displayed).

GENERAL DEVICE DESIGNATOR

(See *device designator*)

GROUP LIBRARY

A *group library* is an area for the storage of programs and files. There are a possible 260 *group libraries* on any system. The last two digits of a *group library idcode* are always zero. You may access your *group library* by prefacing a file or program name with an asterisk.

Examples:

A300, Z400, F900, etc.

HALF DUPLEX

Half duplex is a term used to denote a method of character transmission from your terminal to the system. *Half duplex* means that when a character is typed on your terminal, it is sent to the system and is also printed (or displayed) at your terminal. When the system is in *half duplex* mode, the system does not send the character back to your terminal. If you operate a *half duplex* terminal on a port set for full duplex, every character is displayed twice (once by the terminal and once by the system). To set your port to *half-duplex* mode, type ECHO-OFF.

Example:

```
HHEELL--AA112233,,PPAASS
8-15-75          PORT 23
EECCHH--OOFFFF      (characters then cease being displayed twice)
```

IDCODE

An *idcode* is a letter followed by three digits that serves to identify users of a system. The letter may be A through Z; the three digits may be 000 through 999.

Examples:

A123, B400, A000, Z999, etc.

Each user of the system has an *idcode* which is used to log onto the system. Each *idcode* can have a password, some number of minutes allowed for system connection, a number of disc *blocks* allowed for storage of programs and files, the capability to access non-sharable devices, and capabilities for Program/File Access (PFA), File Create/Purge (FCP), and Multiple Write Access (MWA).

JOB FUNCTION DESIGNATOR

A *job function designator* is a *device designator* which refers to one of the logical data paths between the ACCESS system and a host system. There are five *job function designators*: JM, JI, JP, JL, and JT (job message, job inquiry, job punch, job lister, and job transmitter). *Job function designators* are used as *ASCII files* for Remote Job Entry (RJE) data communications.

LIBRARY

A *library* is an area of storage for programs and files. *Libraries* are specified by the *idcode* which 'owns' the entries. There are three levels of *libraries* on the system: your personal *library, group libraries* (one of which is your *group library*), and the *system library*. Files and programs which reside in *libraries* other than your own may be referenced in several ways. Your *group library* can be referenced by preceding names with an asterisk (*). The *system library* can be referenced by preceding names with a dollar sign ($). Other *libraries* can be accessed by following a name with a period (.) and the *idcode* which 'owns' the *library*.

LIBRARY NAME

A *file name* or *program name*.

NON-SHARABLE DEVICE

A *non-sharable device* is any peripheral device attached to the system which can only be utilized by one user at a time. *Non-sharable devices* include magnetic tape drives, line printers, card readers, paper tape readers, and paper tape punches. In addition, all of the *job function designators* are *non-sharable devices*.

OUT= FILE NAME

The **OUT= file name** construct can be used with certain commands. In commands followed by parameters, it replaces the hyphen. In commands without parameters, it simply follows the command. It causes the output of the particular command to be sent to the ASCII file specified by *file name*. User commands that can use the **OUT= file name** construct are RUN, EXECUTE, LIST, PUNCH, CATALOG, GROUP, LIBRARY, and DEVICE. The three special operator commands used from A000 (DUMP, DIRECTORY, and REPORT) are also allowed use of the **OUT= file name** construct.

Examples:

```
RUN*OUT=LPRTR*50
CAT*OUT=PUNCH*
DIRECTORY*OUT=MAGT*L402
```

PROGRAM NAME

A *program name* is a symbol composed of one to six letters or digits. It is used to identify a specific program in a *library*.

Examples:

PROG, PR1234, TABLEP, etc.

PROGRAM REFERENCE

A *program reference* is an optionally qualified ($, *, .idcode) *program name* used to identify a specific program in a *library*. A *program reference* may refer to a *group library* program by using an asterisk (*) prefix. A *program reference* may refer to a *system library* program by using a dollar sign ($) prefix. Following a name with a period (.) and *idcode* indicates that the named program is to be found in the library of that *idcode*.

Examples:

$PROG, *GRPROG, MYPROG, 5HP298.A123

RECORD

A *record* is the logical unit of storage on the system. All files on the system are made up of one or more *records*. The length of a *record* is its *record length*.

RECORD LENGTH

Record length is the number of words contained in a *record*. BASIC formatted files have *record lengths* of 64 to 256 words and each record occupies one disc *block*. *ASCII disc files* have *record lengths* of from 1 to 255 words and several *records* may occupy one disc *block*. *Non-sharable devices* have *record lengths* of from 1 word to the maximum number of words allowed by the system. A DEVICE command lists the maximum *record length* available for *non-sharable devices*.

SPECIFIC DEVICE DESIGNATOR

(See device designator.)

STATEMENT NUMBER

A *statement number* is an integer in the range 1 to 9999 inclusive. Each statement in a program is preceded by a unique *statement number*. It is used to indicate the proper order of the statement relative to other statements in the program.

SYSTEM LIBRARY

The *system library* is an area for the storage of programs and files. There is one system library per system and its *idcode* is A000. The system library is the personal library of *idcode* A000, but entries within it may be referenced by other *idcodes* by prefacing a name with a dollar sign ($). The *system library* is also the *group library* for *idcodes* A000 through A099.

WORK SPACE

Your *work space* is an area where your current program is stored. When statements are entered at your terminal, they become part of your *work space*. When you GET a program, it is brought into your *work space*. A *work space* may have a name associated with it through use of the GET or NAME command. When a program is saved, that name becomes the name of the program in your *library*. The SCRATCH command clears your *work space* and its name.

COMMAND DESCRIPTIONS

The following pages describe each of the user commands available on the system. Each command description gives the general form of the command, a description of command operation, and any optional parameters.

The description for each command is followed by one or more examples. The descriptions use the terms defined in the previous pages. Whenever terms appear they are printed in *italics*. Terms that are defined locally within the command description itself are emphasized by printing them in ***bold italics***. Optional parameters are enclosed in square brackets ([optional parameters]).

APPEND Command

General Form:

APPEND-*program reference*

The APPEND command retrieves the named program from the appropriate *library* and appends it to the current contents of your *work space*.

The lowest *statement number* of the appended program must be greater than the highest *statement number* of the current program. You can append any program in your library except programs that have been saved with the CSAVE command or that contain the COM statement.

You can append UNRESTRICTED or PROTECTED programs from your *group library* or the *system library* or from *libraries* with the PFA (Program/File Access) capability. If a PROTECTED program is appended from another *library*, the resulting program in your *work space* will also be PROTECTED. Programs that are LOCKED or PRIVATE can be appended only by their owners.

Examples:

```
APP-$PUBLIC
APP-HIS.B904
APP-MINE
APP-*GROUP
```

BYE Command

General Form:

BYE

The BYE command is used to log off of the system.

Entry of the BYE command ends the current session. The system will respond by printing the number of minutes that you have used in the current session and will also update the total time used by your *idcode*. If you are using a modem, telephone connection is broken.

Example:

```
BYE
0009 MINUTES OF TERMINAL TIME
```

CATALOG, GROUP, and LIBRARY Commands

General Forms:

CATALOG [***-library name***]
GROUP [***-library name***]
LIBRARY [***-library name***]

The CATALOG, GROUP, and LIBRARY commands are used to obtain alphabetical listings of programs and files stored on the system.

If the *library name* parameter is used, the listing begins with the first program or file with a name that is equal to or greater than the parameter.

The CATALOG command lists the names of all programs and files stored in your *library*. The GROUP command lists the names of non-private programs and files stored in your *group library*. The LIBRARY command lists the names of non-private programs and files in the *system library*.

The hyphen in the command may be replaced by the **OUT= file name** construct.

Programs, *BASIC formatted files,* and *ASCII disc files* are listed in the following format:

Program or file name; ***Program or file description; Access restriction; length; record***

All other *ASCII files* are listed in the following format:

File name; A; ***Access restriction;*** *specific* or *general device designator;* ***record***

A *program or file descriptor* is one of the following characters:

A — *ASCII file*
F — *BASIC formatted file* with SWA (Single Write Access)
M — *BASIC formatted file* with MWA (Multiple Write Access)
C — CSAVED program
blank — SAVED program

An *access restriction* is one of the following characters:

U — UNRESTRICTED
P — PROTECTED
L — LOCKED
blank — PRIVATE

Length is the number of *blocks* required to store the program or file. ***Record*** is blank for programs and for *BASIC formatted files* having a *record length* of 256 words. For *ASCII files,* ***record*** is the *record length* in words.

Examples:

```
CAT*OUT=DISC*
CAT*OUT=LPR*T
GRO
```

```
LIB-P
 NAME      LENGTH  RECORD   NAME      LENGTH  RECORD   NAME      LENGTH  RECORD
P1       AL    10     63   P2       AL    10     63   PAY      CU     2
PPØ      AL   PPØ     64   PRIN      U     1          PTTTY    CU     2
PZØ75     U     4          PZ125     U     6          PZ325     U    15
RATES    CU     2          SCOOP    CU     1          SCR      FL   1ØØ
SNLIST   FL   1ØØ          STATS    CU     8          T        FL    2Ø
TESTER    U     1          TRADER    L    13          TRADES    L    29
TSPØØ1   CL     8          TSPØØ2   CL    25          TSPØØ3   CL    25
TSPØØ4   CL     9          TSPØØ5   CL    14          TSPØØ6   CL    23
TSPØØ7   CL    15          TSPØØ8   CL    17          TSPØØ9   CL     8
TSPØ1Ø   CL    21          TSPØ6A   CL    21          TSPØ6B   CL    1Ø
TSPHSP    L     8          TSPMSG   FL    36          TSPUØ1    L    17
TSPUØ2    L     8          UWBSP1   CU    14          XCORUN   CU    26
```

CREATE Command

General Form:

CREATE-*file name, file length* [**, *record length***]

The CREATE command builds a *BASIC formatted file* on disc. *End-Of-File* (EOF) marks are written into each record. One disc *block* per *record* is used regardless of the *record length*. The *file length* may be limited by the system configuration and the disc space allotted to your *idcode*. The file is created in the LOCKED state with SWA (Single Write Access) access. You can create files in your own *library* only, so the *file name* cannot be qualified.

Examples:

```
CRE-FILEA, 48
CRE-MYFILE, 123, 128
```

CSAVE Command

(Refer to SAVE)

DELETE Command

General Form:

DELETE-beginning *statement number*

or

DELETE-beginning *statement number*, ending *statement number*

The DELETE command erases all statements in your *work space* between and including the specified statements.

If both *statement numbers* are the same then only the specified statement is deleted. If the ending *statement number* is given, it must not be less than the beginning *statement number*. If no ending *statement number* is given, the delete operation continues through the end of the program. DEL-1 has the same effect as the SCRATCH command except that your *work space* retains the *program name*.

Examples:

```
DEL-27
DEL-27,50
DEL-27,27
```

DEVICE Command

General Form:

DEVICE

The DEVICE command lists the *specific device designators* for the *non-sharable devices* that are available to your *idcode* in the following format:

DEVICE DESIGNATOR	MAXIMUM RECORD SIZE	STATUS

Maximum record size is the largest *record size*, in words, specifiable for the device in a FILE command. The Status field may be blank, indicating that the device is available for use; may contain the word BUSY indicating that the device is being used by another user; or may contain the characters N/A indicating that the system operator has removed the device from the system or assigned it for exclusive use by another *idcode* or RJE.

The **OUT= file name** construct may be used with the DEVICE command.

Example:

```
DEV
```

DEVICE DESIGNATOR	MAXIMUM RECORD SIZE	STATUS
CR0	40	
CR1	40	
LP0	66	BUSY
LP1	66	
JM0	60	
JP0	41	
JL0	67	
JT0	40	
PR0	64	N/A

ECHO Command

General Form:

ECHO-ON

or

ECHO-OFF

The ECHO command allows use of *half-duplex* terminals.

Users with a *half-duplex* terminal must first log on, then type the ECHO-OFF command.

ECHO-ON returns a user to the *full-duplex* mode.

Examples:

```
ECH-OFF
ECH-ON
```

EXECUTE Command

General Form:

EXECUTE-*program reference*

The EXECUTE command clears your *work space* of any previous program, brings the specified program into your *work space*, and begins executing the program.

Execution always begins at the first program statement. When the program terminates, or any chained-to programs terminate, the *work space* is automatically cleared. You can always execute your own programs. Programs that are UNRESTRICTED, LOCKED, or PROTECTED and are saved in the *system library*, your *group library*, or in any *library* having the PFA (Program/File Access) capability can also be executed. You cannot execute PRIVATE programs saved in libraries other than your own.

The **OUT= file name** construct can be used in place of the hyphen in the EXECUTE command.

Examples:

```
EXE-MYPROG
EXE-HERPRG.B456
EXE*OUT=LPR**GROUP
EXE-$SYSTM
```

FILE Command

General Form:

FILE-*file name, device designator* [**, *record length***]

or

FILE-*file name*, DS, *file length* [**, *record length***]

The FILE command creates an *ASCII file*.

The first general form creates an *ASCII file* by merely equating *file name* with the *specific or general device designator* for a *non-sharable device*; no disc space is allocated in this case. The *device designator* must be configured into the system and your *idcode* must be endowed with the capability to access the general device class. Additionally, the device (if *device designator* is specific) or a member of the device class (if *device designator* is general) must be currently available to your *idcode*. (The system operator can dynamically make specific devices unavailable or assign them exclusively to RJE or to a particular *idcode*). *Record length*, if specified, is an integer between 1 and a device dependent maximum determined by the system configuration. A list of the specific devices available to your *idcode*, with their maximum *record length* and current status, can be obtained with the DEVICE command.

If a *general device designator* is specified and the optional *record length* parameter is supplied, there must exist at least one device of that type, with a maximum *record length* greater than or equal to that specified, which is currently available to your *idcode*.

ASCII files built with the FILE command are not opened until they are referenced in a program. At that time, if a maximum *record length* was not specified for the *ASCII file*, the file will assume the maximum *record length* of the *non-sharable device*. If the file is associated with a *general device designator* the system will assign the lowest numbered specific device of that type that has a sufficient *record length* and is available to your *idcode*

The second general form builds an ASCII disc file. *File length* specifies the number of disc *blocks* to be allocated and may be limited by the system configuration and the amount of disc space allotted to your *idcode*. *End-Of-File* (EOF) marks are written over the entire allocated disc area. *Record length* sets an upper limit on the number of words per record and must be an integer between 1 and 255 inclusive; 63 words is the default. Records exceeding this maximum are truncated on the right, shorter records occupy only the space actually required (ASCII data is packed 2 bytes per word and records must begin on word boundaries, therefore the byte following an odd-length record is unused). A word indicating the *record length* precedes each record. Records cannot cross *block* boundaries; if the number of words remaining in a *block* is less than *record length* plus 1, the entire record is written in the succeeding *block*.

All ASCII files are created LOCKED with Single Write Access (SWA).

Examples:

```
FIL-LIST,LP,120
FILE-FASTLP,LP2
FILE-MYFILE,DS,100
FIL-CDMAST,DS,5000,40
```

GET Command

General Form:

GET-*program reference*

The GET command clears your *work space* of any previous program and brings the specified program into your *work space*.

Programs that are PRIVATE or LOCKED can only be accessed with the GET command by their owner. Programs that are PROTECTED or UNRESTRICTED residing in a *library* with PFA (Program/File Access) capability or in the *system library* or your *group library* can also be accessed with the GET command.

Examples:

```
GET-MYPROG
GET-HISPRG.C119
GET-$REC
GET-*ACT
```

GROUP Command

(Refer to CATALOG)

HELLO Command

General Form:

HELLO-*idcode,password* [, ***terminal type***]

The HELLO command is used to log on to the system. Your *idcode* and ***password*** are assigned by the system operator. The ***terminal type*** tells the system what type of terminal you are using. ***Terminal type*** is specified as one numeric digit as follows:

0 HP 2600A, HP 2640A, HP 2749A, or IBM 2741 (default)

1 Execuport 300 or Texas Instruments Silent 700

2 ASR 37

3 HP 2762A/B, TermiNet 300, or TermiNet 1200

4 Memorex 1240

Failure to specify the correct ***terminal type*** can result in a loss of data. If ***terminal type*** is omitted the system assumes the terminal is type 0.

Your ***password*** consists of from 0 to 6 printing or non-printing characters. Entering your password properly validates access to the system. Use of non-printing characters allows a degree of security in that other users cannot see your password.

When selecting a ***password*** you should be aware that certain non-printing characters cause undesirable effects on certain terminals. The system strips line feeds (control-J), nulls (control-@), control-H, carriage return (control-M), rubout (control-_), X-OFF (control-S), and control-X. Some examples of non-printing characters which have terminal dependent meanings are ESCAPE (initiates control sequences on 2640's and TermiNets), control-L (effects a page eject on TermiNets), control-E (on 2640's echo a control-E with a control-F which the system accepts as if typed), etc. Note that a BEL (control-G) rings a bell on most terminals. Since it is audible to others it would not add to the security of your password. You should consider the effect of control characters on your particular terminal when selecting a ***password***.

Examples:

```
HEL-C900,M
HEL-J106,GEO,1
```

KEY Command

General Form:

KEY

The KEY command tells the system that the following input will be from your terminal keyboard. It is used only after a TAPE (paper tape input) operation is complete. It causes error messages suppressed by the TAPE command to be output to the terminal.

Any valid command has the same effect as KEY. Commands substituted for KEY in this manner are not executed if diagnostic messages (indicating syntax errors in BASIC statements) were generated during tape input.

Example:

```
KEY
```

LENGTH Command

General Form:

LENGTH

The LENGTH command prints the number of words in the program currently in your *work space*, followed by the number of blocks needed to save the program. The total disc space used and allocated to your *idcode* is also printed.

Example:

```
LEN
00151 WORDS=01 RECORDS. 00201 RECORDS USED OF  65000 PERMITTED.
```

LIBRARY Command

(Refer to CATALOG)

LIST Command

General Form:

LIST

or

LIST-P

or

LIST-[beginning *statement number*] [, ending *statement number*] [, P]

The LIST command produces a listing of statements in your *work space*, in *statement number* order. Beginning and/or ending *statement numbers* can be specified to obtain a partial listing. If your *work space* has a *program name*, that *program name* precedes the listing.

The letter "P" may be used to produce page formatted output. Fifty-six lines of output are generated per "page unit" together with blank lines to space the listing for cutting into 11-inch sheets for binding or filing.

Listings may be terminated with the BREAK key. PROTECTED programs obtained from other *libraries* cannot be listed.

The **OUT=file name** construct may be used in place of the hyphen. When *OUT=*file name** is used the P parameter is ignored.

Examples:

```
LIS
LIS*OUT=PRTR*
LIS*OUT=HOLD*100,200
LIS-22,55
LIS-P
LIS-400,P
LIS-50,P
LIS-100,400
```

LOCK Command

General Form:

LOCK-*library name*

The LOCK command places the named program or file in the LOCKED state.

Examples:

```
LOC-MYPROG
LOC-MYFILE
```

MESSAGE Command

General Form:

MESSAGE-*character string*

The MESSAGE command sends a *character string* preceded by your port number to the system operator.

The *character string* can be up to 70 characters long. Longer strings are truncated on the right. All printing characters (including quotes, commas, blanks, etc.) are accepted; except for BEL (control-G), non-printing characters are stripped.

If the system operator's message storage area is full, the message:

```
CONSOLE BUSY
```

will be printed on your terminal, indicating that the message has not been sent and should be entered again.

Example:

```
MES-PLEASE MOUNT TAPE GT476 ON DRIVE 3
```

MWA Command

General Form:

MWA-*file name*

The MWA command places the file specified by *file name* in the MWA (multiple write access) state. Your *idcode* must be endowed with the MWA capability. See the ASSIGN and FILES statement discussions in Section XI for a description of Multiple Write Access.

Example:

```
MWA-MYFILE
```

NAME Command

General Form:

NAME [***-program name***]

The NAME command assigns a name to the program currently in your *work space*.

If NAME is entered with no *program name*, no name is assigned to your *work space*. If a name was previously assigned, it is deleted.

Examples:

```
NAME-PROGR1
NAM-ADDER
NAM-MYPROG
NAM
```

PRIVATE Command

General Form:

PRIVATE-*library name*

The PRIVATE command places the named program or file in the **PRIVATE** state. Refer to Section VIII for a discussion of program and file states.

Examples:

```
PRI-MYPROG
PRI-MYFILE
```

PROTECT Command

General Form:

PROTECT-*library name*

The PROTECT command places the named program or file in the **PROTECTED** state. Refer to Section VIII for a discussion of program and file states.

Examples:

```
PRO-MYPROG
PRO-MYFILE
```

PUNCH Command

General Form:

PUNCH

or

PUNCH-P

or

PUNCH-beginning *statement number* [, P]

or

PUNCH-[beginning *statement number*][, ending *statement number*][, P]

or

PUNCH-, ending *statement number* [, P]

The PUNCH command punches the current program onto paper tape if your terminal has a paper tape punch. The program is punched in *statement number* order. Starting and/or ending *statement numbers* can be specified to obtain a tape containing a partial program. In addition, the *program name* and leading and trailing feed holes are also punched. The program is listed on your terminal as it is punched. If your terminal does not have a punch, only a listing will be generated.

The terminal punch must be turned on after the punch command is typed, but before the trailing carriage return is typed.

An X-OFF, carriage return, and line feed are added at the end of each line to enable other BASIC programs to read the paper tape as data or to allow reentry of the program tape. The X-OFF character gives the system control over reading the tape by turning off the paper tape reader while the system processes the line just read. The system sends an X-ON character to the reader when it is ready to read the next line. The X-ON character instructs the reader to continue reading. (See the TAPE command.)

The letter "P" may be used to produce page formatted output. Fifty-six lines of output are punched per "page unit" together with linefeeds to space the output for later listing. This allows for cutting listings into 11-inch sheets for binding or filing. The "P" option will not affect the tape being punched.

Punching may be terminated with the BREAK key. PROTECTED programs obtained from other *libraries* cannot be punched or listed.

The *OUT=*file name** form may be used to divert the output to an ASCII file. When *OUT=*file name** is used the "P" parameter is ignored. If the punching is directed to an ASCII file, that file must be a PP device.

Examples:

```
PUN
PUN*OUT=TAPE*
PUT*OUT=OUTBUF*100,200
PUN-22,55
PUN-P
PUN-,400,P
PUN-50,P
PUN-,400
```

PURGE Command

General Form:

PURGE-*library name*

The PURGE command deletes the specified program or file from your *library*. It will not delete the copy of a program in your *work space*. A purged program or file is recoverable only if another copy exists in your *work space*.

A file may not be purged while it is being accessed by another user. The PURGE command can be used to dissociate *non-sharable devices* from the name used in a previous FILE command. If the file is an *ASCII disc file* or *BASIC formatted file*, the file space that it occupied is returned to the system.

Examples:

```
PUR-PROG12
PUR-FILE09
```

RENUMBER Command

General Form:

RENUMBER

or

RENUMBER-*statement number*

or

RENUMBER-*statement number, interval*

or

RENUMBER-*statement number, interval,* beginning *statement number*

or

RENUMBER-*statement number, interval,* beginning *statement number,* ending *statement number*

The RENUMBER command is used to renumber statements in your *work space*. The initial *statement number* specifies what the number of the first affected statement should be. ***Interval*** is the difference between successive *statement numbers*. Starting and ending *statement numbers* refer to the old *statement numbers* at which the renumbering is to begin and end. *Statement numbers* referenced in GOTO, GOSUB, IF . . . THEN, RESTORE, CONVERT, and PRINT USING statements are automatically replaced with the appropriate new number.

If the initial *statement number* is not given, the first *statement number* of the renumbered program will be 10. If the beginning *statement number* is not given, the renumbering will begin with the first statement of the program. If the ending *statement number* is not given the renumbering will continue to the end of the program. If both beginning and ending *statement numbers* are absent then the entire program will be renumbered.

If no ***interval*** is specified then new numbers will be in increments of 10.

If all parameters are omitted then the entire program is renumbered with the first statement numbered 10, at intervals of 10. RENUMBER cannot be used to change the order of statements. If any command parameter is omitted then all of the parameters following it must also be omitted.

Examples:

```
REN
REN-100
REN-10,1
REN-20,50,100
REN-10,10,50,100
```

RUN Command

General Form:

RUN

or

RUN-*statement number*

The RUN command starts execution of the current program in your *work space* (do not confuse it with the EXECUTE command). Execution normally starts with the first statement in a program. When a *statement number* is given, execution begins at the specified *statement number* or the next highest *statement number* if that specified statement does not exist.

Note that when the RUN-*statement number* form of the command is used, all statements before the specified statement will be skipped. Variables defined in skipped statements will be undefined and cannot be referenced until they are defined in an assignment, INPUT, ENTER, READ, or LINPUT statement. FILES and DIM statements are executed before any other statements regardless of where they appear in a program. They are always executed.

A running program may be terminated with the BREAK key. If you were allowed to GET a program from another *idcode*, you are allowed to RUN it.

The *OUT= *file name** construct may be used with the RUN command.

SAVE and CSAVE Commands

General Form:

SAVE

or

CSAVE

The SAVE command is used to save a copy of the current program. A copy of the program is transferred from your *work space* to your *library*. A program must have a name assigned to it before it can be saved (see the NAME command).

Example:

```
SAV
```

The CSAVE command is also used to save a copy of your current *work space*. The version saved by the CSAVE command will begin execution slightly faster than a SAVED program. This is especially important with large programs that do a lot of chaining.

Example:

```
CSA
```

A PROTECTED program from another user's *library* may not be SAVED nor CSAVED.

SCRATCH Command

General Form:

SCRATCH

The SCRATCH command deletes the entire current program including the *program name* from your *work space*. Scratched programs are lost unless they have been saved elsewhere.

Example:

```
SCR
```

SWA Command

General Form:

SWA-*filename*

The SWA (Single Write Access) command removes the named file from the MWA (Multiple Write Access) state. The file may now only be written to by the first user to access it while that user has the file opened. The single write access state is the default state for files. The SWA command is ignored if your account does not have the MWA capability or if the file is already in the SWA state.

Example:

```
SWA-MYFILE
```

TAPE Command

General Form:

TAPE

The TAPE command tells the system that the following input (a group of BASIC statements) is from the terminal's paper tape reader.

TAPE suppresses any diagnostic messages which are generated by input errors, as well as the automatic (*linefeed*) after (*return*). The KEY command or any other command, causes any diagnostic messages that have been generated to be output to your terminal, ending the TAPE mode.

The tape reader on your terminal must be turned on before typing the (*return*) after the TAPE command, since the system will respond immediately to this command with an X-ON character which turns on the tape reader and initiates reading. Refer to the PUNCH command.

Example:

```
TAP
```

TIME Command

General Form:

TIME

The TIME command responds with one line containing your *idcode*, port number, time used since log-on, total time used for the *idcode*, and maximum time permitted for that *idcode*. All times are expressed in minutes. The output for the TIME command is in the form:

idcode ON PORT # *port number* FOR *time* MIN. *total time* MIN USED OF *max time permitted*.

Time used by each *idcode* is recorded automatically.

Example:

```
TIM
A000 ON PORT # 05 FOR 00025 MIN. 00125 MIN USED OF 65000 PERMITTED.
```

UNRESTRICT Command

General Form:

UNRESTRICT-*library name*

The UNRESTRICT command places the named program or file in the UNRESTRICTED state.

Examples:

```
UNR-MYPROG
UNR-MYFILE
```

BASIC LANGUAGE REFERENCE

SECTION XI

INTRODUCTION

This section contains descriptions of the statements and functions used in programming on the 2000 Access system. A separate heading is used for each statement and function. The headings are arranged in alphabetical order. Following each heading will be either the description for the function or statement or a reference to where the description occurs. Some descriptions have been grouped together functionally for purposes of discussion. Table 11-1 contains an alphabetic list of all of the statements and functions together with the page where they are have been grouped together functionally for purposes of discussion. Note that statement numbers, while required for all statements are not included in the general form descriptions.

BASIC LANGUAGE TERMS

Each description is made up of the statement or function format, explanatory text, and one or more examples. The format is made up of the statement or function together with any required and optional parameters or arguments. The parameters and arguments are described using a set of global BASIC language constructs or terms. These terms are described in the following paragraphs. When these terms are used in descriptions they will be printed in *italics*. Terms which apply only to specific statements or functions are defined within the statement description. These terms are printed in ***bold italics***.

ARRAY

An *array* is an ordered collection of numbers referenced either simultaneously by the associated *array name* or individually by qualifying the *array name*. The second form of reference is called a *subscripted variable* and the individual value specified is called an *array element*. For instance, D(3,8), S(46), and X(101,3) are all *subscripted variables* referring to values that are *array elements*.

An *array* can be one-dimensional, organized as a column of elements (individually referenced by a *subscripted variable* with a single subscript, the value of which specifies the element's position within the column). An *array* can also be two-dimensional, organized as one or more rows of one or more columns of elements (individually referenced by a *subscripted variable* with two subscripts, the value of the first specifying the row and the value of the second specifying the column which contains the element). Within a program all references to elements of a given *array* must be consistent with that *array's* dimensionality. Two-dimensional *arrays* are rectangular; that is, all rows have the same number of columns.

Array A (one-dimensional) =

4.5
3
102
73

where:

A(1) = 4.5
A(2) = 3
A(3) = 102
A(4) = 73

Array B (two-dimensional) =

59	3	96
1	5	32
12	95	2

where:

B(1,1) = 59
B(1,3) = 96
B(2,2) = 5
B(3,1) = 12
B(3,3) = 2

Each *array* has a size, defined as the number of elements it contains. The size of a one-dimensional *array* is simply the number of elements in its single column. The size of a two-dimensional *array* is the product of the number of rows and columns. The size of an *array* can be specified in a DIM statement or COM statement; as few as 1 or as many as 5000 elements can be specified in this way.

Example:

```
DIM M(32), F(20,10), X(500)
```

Array M has 32 elements; Array F has 20 rows and 10 columns or 200 elements; Array X has 500 elements.

If not specified, the system will assign a size of 10 to a one-dimensional *array* or a size of 100 (10 rows by 10 columns) to a two dimensional *array*. Each *array element* occupies 2 words of user work space. Although the definitions of the language allow each *array* in a program to have as many as 5000 elements, the capacity of the Access system imposes lower limits in most cases. The available user work space can contain slightly more than 5000 array elements. Thus one *array* of maximum size is possible. However, this space must be shared with the program itself, other *arrays, numeric simple variables, string simple variables,* buffer space for files, etc. Even a program using only a single *array* and few or none of the above items must be restricted to a small number of statements in order to leave enough space for 5000 elements.

In general an *array* has no attributes beyond those discussed above. A program can treat a one-dimensional *array* as the mathematical entity referred to as a column vector and can treat a two-dimensional *array* as the mathematical entity referred to as a matrix. The MAT statement of BASIC includes several forms which operate on *arrays* according to the rules of vector and matrix arithmetic. Most forms of the MAT statement perform functions on *arrays* which are useful for non-matrix purposes as well.

ARRAY ELEMENT

(Refer to *array*).

ARRAY NAME

An *array name* is a single alphabetic character (A through Z) used to reference a collection of numbers called an *array*. Within a program a given *array name* will always refer to the same unique *array*. However, the form of an *array name* is also one of the allowable forms of a *numeric simple variable.* It is permissible for the same symbol (letter) to be used both as an *array name* and as a *numeric simple variable* within the same program. In each appearance of the symbol, its context determines which of the two names it represents.

Example:

```
110 PRINT K(3,2) , P(201) , K
```

In this example, there is no confusion between the *numeric simple variable* K and the *array name* K because the *array* is subscripted.

CHARACTER

A *character* is any member of the ASCII character set (refer to Appendix A). Most terminals do not provide a means to enter all of the 128 characters defined by ASCII. In addition the system strips the following characters from paper tape reader or terminal input: null, control-H, linefeed, carriage return, X-OFF, control-X, and rubout. Although within the Access system each character occupies an 8-bit field (one half of a word), the most significant bit of each character received from a terminal or paper tape reader is forced to 0, since this bit is used only for parity-checking rather than character differentiation. However, any of the 256 possible 8-bit configurations can be generated internally if needed. String assignments preserve all 8 bits of each character and string comparisons treat each of the 256 different character values as unique.

CONSTANT

A *constant* is either a *numeric constant*, optionally preceded by a plus sign (+) or minus sign (−), or a *literal string*.

Example:

−3.2, 1.59 E3, "ABC", and "ST" '32 '65 are all constants

DESTINATION STRING

A *destination string* is a *string variable* used in a context where it is assigned a new value. BASIC includes a number of constructs which assign a value to a *string variable* (LET statement, LINPUT statement, etc.) In this discussion the term "assign string" will refer to an ordered collection of characters supplied for assignment, independent of the origin of this string. The adjective "current" will refer to the value and length of the string referenced by a *string simple variable* prior to the assignment of the assign string.

Examples:

```
10 LET A$ = "ABC"
20 INPUT A$
30 CONVERT N TO A$(3)
40 SYSTEM A$(3,80), "TIM"
```

In these examples, A$ is always a *destination string*.

If the *destination string* is a *string simple variable,* its entire current value is replaced by the value of the assign string and its *logical length* is set to the assign string's length (an error results if this exceeds the *physical length* of the *string simple variable*). This form discards the current string value and replaces it with the assign string's value.

Example:

If A$ currently has the value "ABC" and the statement 22 A$= "EFGH" is executed, A$ will then have the value "EFGH" and a *logical length* of 4.

If the *destination string* is a *string variable* with a single *substring designator* then an initial portion of its current value is retained and the remainder is replaced by the value of the assign string. Replacement begins at the character position specified by the *substring designator;* an error results if the specified character position exceeds the current *logical length* + 1. Otherwise this would incorporate one or more character positions into the new string value without defining their contents. The *logical length* is set to the specified character position + assign string length − 1. An error results if this exceeds the *physical length* of the *string simple variable.* This form appends the assign string value onto an initial portion of the current string value.

Example:

If A$ currently has the value "ABC" and the statement 33 A$(3)= "EFGH" is executed, A$ will then have the value "ABEFGH" and a logical length of 6. Attempting to execute the statement 44 A$(8)= "EFGH" would result in an error because character position 7 does not exist in the string "ABEFGH".

If the *destination string* is a *string variable* with a double *substring designator* then all of its current value is retained except the range of character positions specified to be replaced. Replacement begins at the character position specified by the first value of the *substring designator* and continues through the character position specified by the second value. An error results if the first character position exceeds the current length + 1. The number of characters replaced is always equal to the second value − first value + 1. If the assign string value contains more characters than needed, only its initial portion is used (i.e., it is "truncated from the right"). If it contains too few characters then enough blanks are appended to achieve the required length (i.e., the assign string value is "blank-padded on the right"). In the special case of a null substring designator the second value − first value + 1 = 0, and no replacement occurs.

If the last character position specified by the *substring designator* is greater than the current *logical length*, it becomes the new *logical length*. An error results if the new *logical length* exceeds the *physical length* of the *string simple variable* of the *destination string*. Otherwise the characters of the current value which follow the replaced portion of the current string value are retained and the *logical length* is not changed. This form either exactly replaces an existing portion of the current string value or appends an exact number of characters onto an initial portion of the current string value. In either case the assign string is modified to obtain the requested number of characters. This modification does not alter the source of the assign string, only the copy used during the replacement.

Examples:

Assume A$= "ABCD", then

```
11 A$(2,3) = "CB" sets A$ = "ACBD"
12 A$(4,8) = "EF" sets A$ = "ACBEF    "
13 A$(1,1) = "GH" sets A$ = "GCBEF    "
14 A$(1,0) = "I" sets A$ = "GCBEF    "
15 A$(10,11) = "J" results in an error because character position 9 does not exist.
```

FILE NAME

A *file name* is a symbol composed of 1 to 6 letters and/or digits. It is used to identify a specific file in a library. In some contexts the name can be optionally qualified ($, *, or idcode). A preceding dollar-sign ($) indicates that the named file is to be found in the system library. A precedng asterisk (*) indicates that the named file is to be found in the group library. Following the name with a period (.) and account number indicates that the named file is to be found in the library of that account number.

Example:

Valid File Names	Invalid File Names
2 FILE	$$TAR
FILE	FILE332
A	FILE #1
TSPMSG	
YRPG1.C302	
*GFILE	
$LUGHA	

FILE NUMBER

A *file number* is a *numeric expression* which evaluates to a number associated with a file in an executing BASIC program. The *numeric expression* is evaluated and rounded (if necessary) to an integer. The result must correspond to a number currently associated with an open file.

FUNCTION REFERENCE

A *function reference* is a numeric-valued function name followed by a parenthesized list of arguments. It is used to request evaluation of the named function and obtain a number as the result. The Access system defines the following numeric-valued functions: ABS, ATN, BRK, COS, EXP, INT, ITM, LEN, LOG, NUM, POS, REC, RND, SGN, SIN, SQR, TAN, TIM, TYP. A function name formed by the characters FN followed by a letter (A through Z) references a user defined function. For each distinct name of this form used in a program there must exist a DEF statement providing its definition. All user defined functions are numeric-valued and require exactly one *numeric expression* as their argument. Their value is the number obtained by evaluating the argument, assigning the result to the parameter mentioned in the associated DEF statement, and then evaluating the *numeric expression* contained in the DEF statement.

Examples:

```
9000 A3 = ATN(33)
9010 P(45) = POS(A$,"11")
9020 PRINT NUM(B$(31,31)),FNA(X)
9030 CONVERT FND (Z+10) TO D$
```

LITERAL STRING

A *literal string* is a textual representation of a string value within a BASIC program. The usual form consists of a pair of double quote marks enclosing an ordered sequence of characters (blank characters are significant). However, some characters cannot be represented in this fashion since most terminals do not use the full ASCII character set, the system uses certain characters for special control purposes and strips them from paper tape reader or terminal input, and the double quote mark is used as the delimiter. By convention an apostrophe followed by a decimal integer (in the range 0 to 255 inclusive) is interpreted as specifying the equivalent internal 8-bit character. Any character can be specified with this convention (some can only be specified in this manner). A general *literal string* can be composed of any mixture of quoted character sequences and individual characters represented with the above convention as long as no two quoted character strings are adjacent. Note that an apostrophe appearing within a quoted character string is simply the character apostrophe.

A *literal string* can contain as few as 0 characters or as many as 255. The former is called the null string and is represented by two adjacent double quote marks (" "). The number of characters accepted in a line generated by a terminal can be as few as 80 or as many as 256 and can vary from terminal to terminal on the same Access system. In practice a *literal string* in a BASIC statement will always be accepted if the entire statement does not exceed 80 characters (all blanks are included in the count but the terminating carriage return is not). The length of a *literal string* is the number of characters it contains (not the number of characters used to represent it) and the value is the ordered collection of these characters.

Examples:

```
"ABC"
"AB" '67 '68
"F" '32 "G"
'10 '13 '72
"HJKL"
" "
"NINE TAILOR'S NEEDLES"
```

LOGICAL LENGTH

The *logical length* is the number of characters in the current string value of a *string simple variable.*

Example:

If A$ = "BEGIN PROGRAM" then the *logical length* of A$ is 13.

LOGICAL SIZE

The *logical size* is the current working size of an *array*. The MAT statement permits changing the size of an *array* during execution. The *logical size* can range between 1 and the *physical size* of the *array*. The *logical size* of a one-dimensional *array* is simply its current working size. The *logical size* of a two-dimensional *array* is the product of its current row size and its current column size.

Example:

```
10 DIM A(32), M(6,8), Z(3000)
```

The *physical sizes* of arrays A, M, and Z are 32, 48, and 3000 respectively. Initially, the *logical size* is set equal to the *physical size*.

NEW DIMENSIONS

New dimensions can be either the number of rows in a one dimensional array enclosed in parentheses or the number of rows and columns in a two dimensional array separated by a comma and enclosed in parentheses. The number of rows or the number of columns may be *numeric expressions* which are evaluated and rounded to integers. The optional new dimensions allows you to respecify the number of rows and columns of an array. These *new dimensions* must be within the limits specified in the original DIM or COM statement or within the default limits set by the system for the array.

In addition the total number of elements (rows for a one dimensional array or rows × columns for a two dimensional array) in the newly dimensioned array cannot exceed the number of elements dimensioned for in the original array.

Example:

```
10 DIM A(10,15), B(20,20)
  .
  .
  .
80 MAT A = CON(10,10)
90 MAT READ B(15,15)
```

Arrays A and B are originally dimensioned 10×15 and 20×20 respectively. Statement 80 changes the working size of array A to 10×10 and sets each element to a one. Statement 90 changes the working size of array B so that only 225 elements will be read from a DATA statement.

NUMBER

A *number* is an approximation to a real number. It is used as the value represented by a *numeric constant*, referenced by a *numeric variable*, or resulting from evaluation of a *numeric expression*. Real numbers are represented with approximately six (seven in a portion of the range) decimal digits of precision. The ranges of real numbers in the Access system are -10^{38} through -10^{-38}, 0, and 10^{-38} through 10^{38}.

NUMERIC CONSTANT

A *numeric constant* is a textual representation of a number. Its value is the closest approximation possible to the real number specified by the form of the *numeric constant*. The simplest forms are an integer (a sequence of one or more digits) and an integer followed by a decimal point (.), optionally followed by an integer. Either of these forms can be followed by an exponent (the letter E followed by a one or two-digit integer in the range 0 to 38 inclusive, which is optionally preceded by a plus sign (+) or minus sign (−)). An exponent has the value of 10 raised to the positive or negative power indicated by the following integer. The number represented by the complete form is the value of the integer or fraction multiplied by the value of the exponent, if present.

Examples:

```
19.796
−6.768 E 23
123
```

NUMERIC EXPRESSION

A *numeric expression* is a combination of operators and *primaries* which can be evaluated to a number. The general form of a *numeric expression* is too complex to explain all at once. The discussion will first describe a simpler form and then proceed to generalize it. The most common form is the 'arithmetic expression', which closely resembles the simple algebraic expression of mathematics. An arithmetic expression can be a sequence of one or more sums, where each sum (except the last or only one) is syntactically separated from its successor by either MIN or MAX. During evaluation these symbols are interpreted as binary operators. Starting with the first sum, each MIN or MAX compares the value of the preceding portion of the *numeric expression* with the value of the following sum and selects the minimum or maximum respectively as its result. The result of the last MIN or MAX is the value of the entire *arithmetic expression*.

A sum is a sequence of one or more 'terms', where each term (except the last or only one) is syntactically separated from its successor by either a '+' or '−'. During evaluation these symbols are interpreted as binary operators specifying addition or subtraction respectively. A term is a sequence of one or more 'factors', where each factor (except the last or only one) is syntactically separated from its successor by either a '*' or '/'. During evaluation these symbols are interpreted as binary operators specifying multiplication or division respectively. Each factor (including the first or only one) can by option have a preceding '+' or '−'. During evaluation these symbols in this context are interpreted as unary operators. The unary '−' specifies negation of the value of its following factor; the unary '+' exists only for syntactic symmetry and has no effect on evaluation. A factor is a sequence of one or more *primaries*, where each *primary* (except the last or only one) is syntactically separated from its successor by either '↑' or '**'. These symbols are alternative representations of the same binary operator, called the exponentiation operator. During evaluation they specify that the value of the *primary* following the operator is used as a power and the value of the preceding portion of the factor is used as a base (e.g., A**B is equivalent to the mathematical notation A^B).

The use of '+' and '−' as both binary and unary operators is traditional in mathematics. Unfortunately it can create confusion in some cases. Consider the following three legal arithmetic expressions:

term − factor term − + factor term + − factor

By definition any term except the last one must be followed by a '+' or '−', while a factor is only optionally preceded by a unary '+' or unary '−'. Thus the proper interpretation of the first case is as 'term − term', where the second term is a simple factor. The second case is also 'term − term', where the second term is a factor preceded by a unary '+' and hence is equivalent to the first case. The third case is 'term + term', where the second term is a factor preceded by a unary '−'. Under the rules of mathematics, subtraction of a number is equivalent to addition of its negative value. The result is that all three of the above cases will evaluate to the same value.

The following diagram can be read from top to bottom as an illustration of how *primaries* and operators combine to form an arithmetic expression, or it can be read from bottom to top as an illustration of how the definition of an arithmetic expression is expanded into its components.

```
   A     * -     B      **     C      +     D      /     E
(primary)     (primary)     (primary)     (primary)     (primary)

     A     *      - B**C      +     D      /     E
  (factor)     (negated factor)     (factor)     (factor)

          A* -B**C   +   D/E
           (term)        (term)

          A* -B**C+D/E
       (arithmetic expression)
```

In any *arithmetic epression* which contains two or more operators, the order of their application during evaluation is important. In BASIC the order of evaluation is implicitly specified by a hierarchy of precedence among the operators. This hierarchy, adopted from the conventional rules of mathematics, is illustrated by the following table:

```
**      ↑     (highest level)
unary -       unary +
*       /
+       -
MIN     MAX     (lowest level)
```

When contemplating two operators which appear at different levels in this table, the operator at the higher level is chosen for execution first. In the arithmetic expression A+B*C the multiplication precedes the addition. When two operators appear at the same level, the operators are executed from left to right. In A+B−C the addition precedes the subtraction. In some cases the implicit order of evaluation is not the one desired. Note that one form of a *primary* is a parenthesized *numeric expression*. Any portion of a *numeric expression* which by itself is also a legal form of *numeric expression* can be enclosed in parentheses. The enclosed portion becomes a *primary*, forcing its evaluation to precede execution of any operators in the surrounding *numeric expression* which might otherwise take precedence. Parentheses can be nested to any depth required to override the implicit order of evaluation.

Example:

```
A+B/C MIN -D*+E**F      is evaluated in the same order as
(A+(B/C)) MIN ((-D)*(+(E**F)))
```

Arithmetic expressions can be combined into a more complex form of *numeric expression* called a 'relational expression'. A relational expression is a sequence of one or more arithmetic expressions (usually two), where each arithmetic expression (except the last or only one) is syntactically separated from its successor by a *relational operator*. During evaluation a *relational operator* is interpreted as a binary operator which produces a value of 'true' or 'false'. The most common use for a relational expression is within an IF statement.

Example:

```
IF A+B <= C+D THEN 300
```

The most general form of *numeric expression* is the 'logical expression'. Three logical operators are available: AND, OR, and NOT. The first two are binary operators used to combine relational expressions into more complex decisions. The result of executing OR is 'true' unless both of its operands have the value 'false'. The result of executing AND is 'false' unless both of its operands have the value 'true'. NOT is a unary operator and has the same precedence as unary '−' and unary '+'. NOT performs a logical negation; its result is 'false' if the value of its operand is 'true' and its result is 'true' if the value of its operand is 'false'.

Example:

```
IF (A+B>C AND X=Y) OR NOT D<= E+2 THEN 45
```

Although logical operators are described in terms of manipulating 'true' and 'false', the values are actually represented within BASIC as numbers. If the result of a logical operator or *relational operator* is 'true', it produces the number 1. If the result is 'false' it produces the number 0. The operand of a logical operator will be interpreted as 'true' if its value is non-zero; it will be interpresed as 'false' if its value is 0. Logical operators can use any *numeric expression* as their operand and both logical operators and *relational operators* can appear within *numeric expressions* enclosed in parentheses and used as primaries of arithmetic expressions. The complete operator hierarchy for *numeric expressions* is the following:

```
**     ↑     (highest)
unary +      unary −      NOT
*     /
+  −
MIN      MAX
<=  <  =  >  >=  <>  #
AND
OR      (lowest)
```

The same rules for order of evaluation given under the discussion of arithmetic expressions apply to the expanded table. Parentheses can be used to override the implicit order of evaluation at any level of a *numeric expression*. Note that A<B<C is not equivalent to A<B AND B<C. The format evaluates A<B to a 1 or 0 and then compares that result to C. The latter evaluates A<B to 'true' or 'false' (1 or 0) and B<C to 'true' or 'false' (1 or 0) and then produces the logical AND of the two results.

Example:

```
A AND B<C+D OR NOT E      is evaluated in the same order as
(A AND (B<(C+D))) OR (NOT E)
```

NUMERIC SIMPLE VARIABLE

A *numeric simple variable* is a single alphabetic character (A through Z) optionally followed by a single digit (0 through 9). It is used to reference a number, which is also referred to as its value. The value can be accessed or altered by several BASIC language constructs. Normally a *numeric simple variable* is uniquely identified by its name and its value is available to any statement within a program. However, a special case exists when a name of this form appears as the parameter of a DEF statement. In this context the value of the name is assigned by a reference to the associated function name and is accessed by use of the parameter name as a *primary* within the numeric expression of the DEF statement. The same name can appear as the parameter of different DEF statements within the same program and also as a *numeric simple variable* in other statements. A parameter is recognized as unique within the context of its own DEF statement; thus all appearances of its name outside of that statement are independent. The values associated with these other appearances are maintained separately and do not interact with the value of the parameter.

Example:

```
A9, P3, G, C, F0, Z2
```

NUMERIC VARIABLE

A *numeric variable* is either a *numeric simple variable* or a *subscripted variable.*

Examples:

```
H2, F(3), M(I,J+2), Z
```

PHYSICAL LENGTH

The *physical length* is the maximum number of characters that the *string value* of a *string simple variable* can contain.

Example:

```
DIM A$(30), P1$(72)
```
(defines the physical lengths of A$ and P1$ as 30 and 72, respectively)

PHYSICAL SIZE

The *physical size* is the maximum size an *array* can attain. This is the same as the size established either explicitly in a DIM statement or COM statement or implicitly in the absence of an explicit specification.

Examples:

```
DIM A(45), B(3,2)
```
(defines the physical size of A, B, and M as 45, 6, and 30, respectively)

```
COM M(30)
```

PRIMARY

A *primary* is a *numeric constant*, a *numeric variable*, a *function reference*, or a *numeric expression* enclosed within parentheses. *Primaries* are used to supply the numbers which are combined and manipulated by the operators of a *numeric expression*.

PROGRAM NAME

A *program name* is a symbol composed of 1 to 6 letters and/or digits. It is used to identify a specific program in a library. In some contexts the name can be optionally qualified ($, *, or .idcode). A preceding dollar-sign ($) indicates that the named program is to be found in the system library. A preceding asterisk (*) indicates that the named program is to be found in the group library. Following the name with a period (.) and idcode indicates that the named program is to be found in the library of that idcode.

Examples:

Valid Program Names		Invalid Program Names
PROG	YRPRG.F932	PROGRAM
B	$DRAG	PROG/2
TDE200	*GPROG	

RECORD NUMBER

A *record number* is a *numeric expression* which evaluates to a number specifying a particular record within a BASIC formatted file. The *numeric expression* is evaluated and rounded (if necessary) to an integer. The result must be greater than or equal to 1 and less than or equal to the number of records in the file.

RELATIONAL OPERATOR

A *relational operator* is one of the symbols:

<	(less than)
<=	(less than or equal)
=	(equal)
>=	(greater than or equal)
>	(greater than)
<>	(unequal)
#	(alternate for <>)

Each symbol is interpreted as a binary operator during execution. It compares the value of the operand to its left with the value of the operand to its right and returns the value 'true' (1) if they satisfy the relation represented or 'false' (0) if they do not. *Relational operators* can be used either to compare the values of two strings (within an IF statement only) or to compare two numbers within a *numeric expression*. Note that the symbol used to test for equality (=) is the same as the symbol used to specify assignment in a LET statement. If this symbol appears more than once within a LET statement, there can be confusion as to which meaning it has (multiple assignment or test for equality). In such cases the rule is that, starting from the beginning of the statement, all appearances of '=' are assignment operators until one of them is followed by a construct which is not a *numeric variable* (parenthesized occurrences are not counted). Any remaining appearances are the *relational operator*.

RETURN VARIABLE

A *return variable* is a *numeric variable* used in the context of a statement which returns status information to a BASIC program. The statement indicates the results of its execution by assigning a value to the *return variable*. The program can then test the result of such statements by accessing the value.

Example:

```
100 ASSIGN FIL1, 3, Z      (Z is a return variable)
```

SOURCE STRING

A *source string* is a *literal string* or a *string variable* used in a context where it supplies a string value. BASIC includes a number of constructs which manipulate a string value (PRINT statement, UPS$ function, etc.). The rules of interpretation for each of these are appropriate specific cases of several general rules. In this discussion the term "accessed string" will refer to the ordered collection of characters supplied by the *source string*. The adjective "current" will refer to the value and length of the string referenced by a *string simple variable* from which an accessed string is taken.

If the *source string* is a *literal string* then the value and length of the accessed string are simply those of the string represented textually. If the *source string* is a *string simple variable* then the value and length of the accessed string are the current value and current logical length of the string referenced by the variable's name. This form returns the entire current string as the accessed string. If the *source string* is a *string variable* with a single *substring designator* then the portion of the current value preceding the specified character position is ignored. The value of the accessed string consists of any remaining characters of the current value and the length is simply the number of them. If the specified character position exceeds the current *logical length*, then the accessed string is the null string. This form discards an initial portion of the current string and returns the (possibly null) remainder as the accessed string. The current string is not altered.

If the *source string* is a *string variable* with a double *substring designator* then the specified portion of the current value is extracted. Extraction begins at the character position specified by the first value of the *substring designator* and continues through the character position specified by the second value. If the last character position is greater than the current *logical length*, the non-existent characters are interpreted as blanks. If the first character position is greater than the *logical length* of the current value, the value of the accessed string will consist entirely of blanks. The length of the accessed string is determined by the *substring designator* and is equal to the second value − first value + 1. In the special case of a null string designator, none of the current value is used and the accessed string is a null string. This form returns a string with a predetermined length and a value composed of characters extracted from the current value where possible, but having blanks corresponding to character positions requested beyond the current logical length. The current string is not altered.

Example:

Assume A$ = "ABCDE", then the *source strings* A$, A$(3), A$(1,3), and A$(4,7) equal "ABCDE", "CDE", "ABC", and "DE ", respectively.

STATEMENT NUMBER

A *statement number* is an integer in the range 1 to 9999 inclusive. Each statement in a program is preceded by a unique *statement number*. It is used both to indicate the proper order of the statement relative to other statements in the program and as a label by which other statements can reference its statement during execution.

Example:

```
100 A=B      (100 is the statement number)
```

STRING

A *string* is an ordered collection of characters which taken all together comprise the *string's* value. The number of characters is the *string's* length, which can be between 0 and 255 inclusive. The maximum length of a *string* can be specified in a DIM statement or COM statement. The minimum length of a *string* specified in a DIM or COM statement is one character. If not specified, the system will assign a length of 1 to a *string*. The actual number of characters used in a *string* may be less than the number dimensioned.

STRING EXPRESSION

A *string expression* is one of several BASIC constructs used to supply a string value. The specific interpretation and use depends upon the context in which it appears. Any *source string* can be used as a *string expression*. In addition the two string-valued functions CHR$ and UPS$ can be used.

Examples:

```
"ABC"
"AB" '67
A$
A$(3,6)
CHR$(I)
UPS$(B$)
```

STRING LENGTH

(Refer to *string*)

STRING SIMPLE VARIABLE

A *string simple variable* is a single alphabetic character (A through Z), optionally followed by either 0 or 1, followed by a dollar-sign ($). It is used to reference a string. The value and length of the referenced string can be accessed or altered by several BASIC language constructs.

Examples:

```
A$, P1$, F$, Z0$
```

STRING VALUE

(Refer to *string)*

STRING VARIABLE

A *string variable* is either a *string simple variable* or a *string simple variable* qualified by a *substring designator.*

Examples:

A$(4), Q1$(1,LEN(Q$)), M$(I,J), G0$, D$

SUBSCRIPTED VARIABLE

A *subscripted variable* is an *array name* followed by either one or two subscripts enclosed in parentheses. It is used to reference the value of an *array element.* A subscript is a numeric expression which is evaluated and rounded (if necessary) to an integer. An element of a one-dimensional *array* is referenced by following the *array name* with a single subscript within the parentheses. It must evaluate to an integer from 1 to the *logical size* of the *array.* An element of a two-dimensional *array* is referenced by following the *array name* with two subscripts, separated by a comma, within the parentheses. The first must evaluate to an integer from 1 to the current row size; the second must evaluate to an integer from 1 to the current column size.

Examples:

F(I,J+3), S(M1*F/2), M(32)

SUBSTRING DESIGNATOR

A *substring designator* is a modifier which can follow a *string simple variable* to designate either an initial character position or a range of character positions. The general form is:

(*numeric expression* [,*numeric expression*])

If only one *numeric expression* appears then the designator is single, specifying an initial character position. If both *numeric expressions* appear then the designator is double, the value of the first *numeric expression* specifying the first character position and the value of the second specifying the last character position of a range. Each *numeric expression* is evaluated and rounded (if necessary) to an integer before use. The value of the first (or only) one must be one of the integers 1 through 32767 inclusive (in practice values above 255 have little utility). The number of character positions in the range specified by a double *substring designator* is the second value − first value + 1. Thus the value of the second *numeric expression* is normally greater than or equal to the value of the first, but cannot exceed 32767. As a special case the second value can be 1 less than the first value, specifying a range of 0 character positions (referred to as the null substring designator). An error results if the values specify a negative range (second value − first value + 1 < 0) or a range greater than 255. Other than these restrictions, the legality of a particular *substring designator* depends upon the context of its appearance.

ABS Function

General Form:

ABS (*numeric expression*)

The ABS function is a numeric valued function which returns the absolute value of the *numeric expression*.

The absolute value of a number is that number, irrespective of the sign (positive or negative) of the number. For example, ABS(33) = 33, and ABS(−33) = 33.

Example:

```
20 PRINT ABS(N/D*S)
```

ADVANCE Statement

General Form:

ADVANCE # *file number;skip count,return variable*

The ADVANCE statement causes the pointer for the specified file to be advanced past the number of items specified by the ***skip count***.

The ***skip count*** is a *numeric expression* which, when evaluated and rounded to an integer, specifies the number of items to be skipped. The number of items to be skipped may not be negative. If the statement is executed successfully, the *return variable* is set to 0. If the statement encounters an EOF (End-Of-File mark) before the specified number of data items have been skipped, the *return variable* will be set to the number of items yet to be skipped. If the statement encounters an EOR (End-Of-Record mark) before the specified number of data items have been skipped, skipping continues into the next record. The ADVANCE statement cannot be used on an ASCII file; if attempted, the program will be terminated. Specification of an invalid *file number* will result in an error. (Refer to the FILES statement for an explanation of file number.)

Example:

```
10 ADVANCE #3;5*X+1,Z
```

ASSIGN Statement

General Form:

ASSIGN *file designator,file number,return variable*[,*mask*][,*restriction*]

or

ASSIGN *,*file number*[,*return variable*]

The ASSIGN statement is used to assign a ***file designator*** to a *file number* reserved by the FILES statement and to open the specified file. For example, ASSIGN AFILE, 3, VAR assigns the file name AFILE to the position reserved for the file number 3 in the FILES statement.

If another file is associated with the *file number* used in an ASSIGN statement, then that file is closed. The result of the assign operation is returned in the *return variable*. The ***file designator*** is a *source string* whose value is a *file name*. The *file name* referenced may be a system library file (**$ *name***), a group library file (****name***), or a file resident (saved) in any other user's library (***name.idcode***). To reference a file with ***name.idcode***, that idcode must have the PFA (Program/File Access) capability. The *file number* can be a *numeric expression* which evaluates to the *file number* (1 — 16).

If an asterisk (*) is used in place of the ***file designator***, the file previously associated with the file number is closed. If the file has already been closed, the statement is ignored.

Example:

```
100 ASSIGN *, I, Z
```

After ASSIGN is executed, a value is returned in the *return variable*. The return values and their meanings are given below.

Return Value	Meaning
0	file is available for read and write
1	file is available for read only
2	file is available for read only (it belongs to another user and is protected)
3	file does not exist or is not accessible
4	file number is out of range (it does not correspond to one of the positions reserved by a FILES statement)
5	no buffer space is available for the file
6	file is not available for read or write because of another user's current access
7	specified restrictions not possible
8	file is available for write only

If the value returned is 3, 4, 5, 6, or 7 the file is not opened, any access to the file number causes the program to be terminated with an error. If the returned value is 1 any attempt to print onto the file causes a terminal error. If the returned value is 8, any attempt to read the file causes a terminal error. Other references to the file assigned that file number are legal.

ASSIGN Statement (Cont)

The optional ***mask*** is a *source string* used to encode or decode BASIC formatted file data (refer to Section V). When using a ***mask***, all data read or written to the file will be modified by the ***mask***, making the data unintelligible to programs which use the file without the same ***mask*** used initially. In order to extract data from a file, the same ***mask*** must be used to read the data as was used to print the data to the file. Numeric zeros, data types, EOR (End-Of-Record) marks, and EOF (End-Of-File) marks are not affected. A ***mask*** used with an ASCII file is ignored.

Example:

```
100 ASSIGN FIL1, I, Z, M$
```

The optional ***restriction*** is a two-letter code used to specify any special access restrictions on the file. Restriction codes are as follows:

Code	Meaning
RR	Read and write restriction; no subsequent user can access the file while the file is open.
WR	Write restriction; subsequent users can read from, but not write to the file while the file is open.
NR	No restriction; subsequent users can read and write file data while the file is open. (As long as the file is multiple write access — MWA.)

If the ***restriction*** parameter is omitted, the file is opened with the WR (Write Restriction) restriction. If this fails because another user has write access, NR (No Restriction) is used. Note that for MWA (Multiple Write Access) files, the NR code is always used.

The specified restriction remains in effect as long as the file is open. If another program has opened the file and restricted its use, then the ASSIGN statement will return a value of 6 or 7 in the *return variable* and the file will not be opened.

For a file located in the user's log-on account that is not currently in use, read/write access is granted. ASCII Files will be opened with read-only or write-only access if appropriate for the device. If the file is in use, and it is single write access (SWA), read-only access is granted unless the other user has applied the RR restriction when assigning the file, in which case no access is granted. If the file is in use and it is multiple write access (MWA), read/write access is granted unless the other user has applied the WR restriction when assigning the file, in which case read-only access is granted. If the other user has applied the RR restriction, no access is granted.

ASSIGN Statement (Cont)

For a BASIC formatted or ASCII disc file located in a library other than (outside) the user's log-on account, that is *not* currently in use, the access granted is as follows:

File Status	Static Access
Unrestricted	Read/Write
Protected	Read only
Locked	No access unless the program accessing the file is saved in the same account as the file and the program is locked, in which case Read/Write access is granted.
Private	No access

You may not access an ASCII file located in a library other than your log-on account except for the following two exceptions: 1) ASCII disc files, or 2) Locked ASCII files in the A000 library which you access by a locked program also saved in the A000 library.

For any file currently in use, located in a library other than the account you are using, the access granted is the same as described above, modified by any dynamic restrictions (NR, WR, RR) that the other user may have applied.

A locked or private program in a group library account with the FCP (File Create/Purge) capability which has been executed or chained-to, has static read/write access to locked BASIC formatted files having the PFA (Program/File Access) capability, located in any of the accounts of that group.

In the example below, files are assigned for each file number associated with an * in the FILES statements. A write restriction (WR) on XFILE prevents other users from writing on that file. A read and write restriction (RR) on DFILE prevents other users from having any access to the file. There are no access restrictions (NR) on FF1. The mask "ABZ1" is used to encode the data in file X. File X is closed in line 90. In line 100, AFILE is closed and file CC is opened as file number 1. The zeros in the numeric variables indicate that each file was available for reading and writing when it was opened.

Examples:

```
 10 FILES AFILE,BFILE,*
 20 FILES AA,BB,*,*,*
 30 ASSIGN "XFILE",3,X1,WR
 40 ASSIGN "DFILE",6,D1,RR
 50 ASSIGN "FF1",7,N,NR
 60 LET X$="X"
 70 ASSIGN X$,8,X,"ABZ1"
 80 PRINT X1,D1,N,X
 90 ASSIGN *, 8
100 ASSIGN "CC",1,C1
110 PRINT C1
120 END
RUN
0         0         0         0
0
```

ATN Function

General Form:

ATN (*numeric expression*)

ATN is a numeric-valued function which returns the arctangent of the *numeric expression*.

The value of the function is the angle (in radians) whose tangent is the *numeric expression*.

Example:

```
100 A = ATN(B)
```

BRK Function

General Form:

BRK (*numeric expression*)

The BRK function enables or disables the BREAK key capability of a terminal.

The value of *numeric expression*, when evaluated and rounded to an integer, will have the following effect:

- Value less than zero returns the status of the BREAK capability.
- Value equal to zero disables the BREAK capability.
- Value greater than zero enables the BREAK capability.

BREAK CAPABILITY. For a program running at a terminal, the BREAK key capability of the terminal can be disabled or enabled by execution of the BRK function within the program. At the beginning of program execution, the BREAK capability is enabled (default). If disabled, it remains disabled until program execution is completed; the program terminates because of an execution error; the BREAK command is entered by the system operator; or until the BRK function is executed with an argument greater than zero.

BRK Function (Cont)

Because program execution may be completed before all output is complete, care should be taken when re-enabling the BREAK capability to ensure that program output will not be interrupted and lost. Either an INPUT statement or an ENTER statement can be included in the program just prior to the statement containing the BRK enable function. This will cause the program to pause until output is complete before continuing execution. For example, the following program segment will disable the BREAK capability and print the value of "I" twenty times. On encountering the ENTER statement, the program will pause until printed output is complete before execution continues from statement 30.

Examples:

```
 5 Y=BRK(0)
10 FOR I=1 TO 20
15 PRINT "I=";I
20 NEXT I
25 ENTER 1,A,B
30 Z=BRK(1)
99 END
```

VALUES RETURNED BY BRK FUNCTION. For arguments equal to or greater than zero, the value returned depends on the previous condition of the BREAK capability. This value will be 1 if the capability was previously enabled, or 0 if the capability was previously disabled.

To find the current status of the BREAK capability, enter an argument less than zero. If currently enabled, a 1 is returned. If currently disabled, a 0 is returned.

If a program is in an infinite loop during execution and the BREAK capability is disabled, the system operator can enter a BREAK command to enable the BREAK capability. Once the system operator enables the BREAK capability, the running program may not disable BREAK until program termination.

For terminals connected to the system through telephone lines, a loss of carrier for longer than two seconds causes the user to be disconnected and automatically logged off the system. Similarly, hardwired terminals that drop carrier and/or data set ready signals when turned off cause the user to be automatically logged off the system. In either case, a disabled BREAK capability is returned to the enabled condition.

Examples:

```
935 LET B = BRK(0)
940 Z = BRK(A+M)
945 PRINT BRK(Y)
```

CHAIN Statement

General Form:

CHAIN [***return variable,***]***program designator***[**,***numeric expression***]

The CHAIN statement causes the current program to terminate and the program referenced by ***program designator*** to be executed.

The ***program designator*** is a source string whose value is a *program name*. The *program name* may be a system library program (***$name***), or a program in any other user's library (***name.idcode***). To reference a program with ***name.idcode***, that ***idcode*** (account) must have the PFA (Program/File Access) capability. You may always CHAIN to a program located in the account you are using, regardless of its status (i.e., Locked, Private, Protected, or Unrestricted).

For a program located in a library other than the account you are using, the access granted is as follows:

Program Status	Static Access
Unrestricted	May chain, optionally specifying a statement number.
Protected	May chain, optionally specifying a statement number.
Locked	May chain, but no statement number may be specified unless the current program is saved in the same library as the chained-to-program.
Private	Chain not allowed.

The optional *numeric expression* following the ***program designator*** can be used to define the *statement number* in the destination program where execution will begin. If not specified, the new program will begin execution at the first program statement. The *numeric expression* is evaluated and rounded to the nearest integer to obtain the starting *statement number*.

If the optional *return variable* is provided and chain operation is successful, the *return variable* is set to 0. This value is accessable in the destination program (if the *return variable* is specified as common).

The CHAIN statement can produce the same errors as the GET command. If a chain operation is unsuccessful, the *return variable* will be set to a value of 1, 2, or 3, depending on the type of error. A list of *return variable* meanings is as follows:

Return Value	Meaning
0	Successful
1	Bad statement number specified (less than 1 or greater than 9999)
2	No access permitted to named program
3	Chain not permitted

CHAIN Statement (Cont)

When a *return variable* is used and a chain operation is not successful, execution of the current program will continue with the statement following the CHAIN statement. Unsuccessful chain operations without a *return variable* will cause the current program to be terminated with an error.

If a chain operation is successful, all files in the chaining program are closed, even if they appear in FILES or ASSIGN statements in the destination program. No other user, however, may gain access to files in the chaining program until any FILES statements in the chained-to program are executed. Also, any variables listed in a COM statement are transferred (see the COM statement elsewhere in this section).

Before execution of the destination program can begin, it must be compiled. Programs may be stored in the semi-compiled form to speed execution (see the CSAVE command in Section X).

Examples:

```
 20 CHAIN "PROG2"
 50 CHAIN V$
 97 CHAIN "ABC",A
150 CHAIN "MELVIN",80
200 CHAIN N$,Q+14
230 CHAIN A$,110
240 CHAIN R,"LIBROC.A001",10
```

CHR$ Function

General Form:

CHR$ (*numeric expression*)

The CHR$ function returns a single ASCII character that has the value of *numeric expression*.

Each character in a string is internally represented by one byte consisting of eight bits. These eight bits allow 256 different characters to be represented. The representation of the first 128 of these characters (0 — 127) is in accordance with the ASCII standard. The last 128 characters have no predetermined meaning (have only a positive value from 128 to 255). All 256 characters can be referred to by a decimal number in the range of 0 to 255. Appendix A presents the decimal equivalents for each of the representable characters (i.e., A = 65, B = 66, etc).

The CHR$ function returns a single character (byte) which corresponds to the position in the ASCII code table (Appendix A) for the argument of the function (the *numeric expression*). The *numeric expression* is evaluated and rounded to an integer which should be in the range of from 0 to 255. If the integer is not within this range, then the program terminates with an error. For example, CHR$(65) and CHR$(30 + 35.4) both yield the character "A".

The CHR$ function may only be used on the right side of a string assignment statement; string IF statement; or as a print list item.

Examples:

```
200 PRINT CHR$(2*(1+B))
300 A$(1,1) = CHR$(J)
```

COM Statement

General Form:

COM *common list*

The COM statement lists variables to be passed between programs linked by the CHAIN statement.

A ***common list*** is one or more common elements separated by commas. A common element is a *numeric simple variable, string simple variable, string simple variable* (*length*), *array name* (number of rows), or *array name* (number of rows, number of columns).

Example:

```
100 COM A,B(10,10),B$(200)
```

Several programs may be run sequentially, all accessing and possibly changing data in the common area. Program control must be transferred with the CHAIN statement. Merely getting and running the next program will not preserve the common area. COM statements must be the lowest numbered statements in the program. The transfer of data values between variables in different programs is done according to the order of the variables in the COM statement. For example, if one program has as its first statement

```
10 COM A,B1,C$(10)
```

and a second program has as its first statements

```
12 COM X
14 COM Y,Z$(10)
```

and the first program chains to the second, the current values of variables A,B1, and C$ will be used to initialize the variables X, Y, and Z$ respectively. Note that you are free to use the variable names A,B1 and C$ for a different purpose in the second program.

Array and string variables used in COM statements must be dimensioned in the COM statement and may not appear in DIM statements. The corresponding variables in the chained-to program must also be dimensioned in the COM statement and must be dimensioned to the same values as those in the first program. In order to access a common variable, all preceding common variables must correspond in type and dimension to those in the first program. You need not have the same number of common variables in both programs.

The values of common variables are lost when a program terminates (whether normally due to an END or STOP statement; to an execution error; or to the use of the BREAK key).

Examples:

```
100 COM A,Q0$ (255)
200 COM C,R1,C3
300 COM F(30,12), B$(10), Z(14)
```

CON Function

(Refer to MAT . . . CON)

CONVERT Statement

General Form:

CONVERT *numeric expression* TO *destination string*

or

CONVERT *source string* TO *numeric variable*[, *statement number*]

The CONVERT statement is used to change a *numeric expression* to a string of characters that represent the value of the expression. It may also be used to convert a *source string* to an equivalent numeric value.

NUMERIC TO STRING. When converting numeric values to string values, the conversion is the same as that used by the system to list numeric data (using the LIST command). The output string will not contain embedded blanks.

Example:

```
150 A=10
170 B=15
200 CONVERT A+B TO A$
220 PRINT A$
```

This will result in the string " 25" being printed as three characters as in the normal numeric format of a listing.

STRING TO NUMERIC. When converting strings to numeric values, the string must represent a valid numeric constant. If not, the program will be terminated unless an optional statement number is specified. In this case, an invalid numeric value will transfer control to that statement. Blanks are permitted in the string.

Examples:

```
200 CONVERT P1$(21,30) TO F(7)
300 CONVERT A$ TO X,500
400 CONVERT B$ TO M1
```

COS Function

General Form:

COS (*numeric expression*)

The COS function is a numeric valued function which returns the cosine of the *numeric expression*. The *numeric expression* is interpreted as being in radians. If the absolute value of the *numeric expression* exceeds approximately 102900, your program will be terminated with an error.

Example:

```
500 LET B = COS(3.1415/X)
```

CREATE Statement

General Form:

CREATE *return variable,file designator,file length*[, *record size*]

The CREATE statement is used to create a BASIC formatted file from an executing program.

The CREATE statement can be used only to create a BASIC formatted file and cannot be used to create an ASCII file. The *return variable* is set to a value which indicates the result of the execution of the statement. The values which may be returned and their meanings are as follows:

Return Value	Meaning
0	The file was created successfully
1	A file already exists with the same name
2	Bad file name, no such account, invalid access, bad file length, or bad record size
3	No space in the account
4	No space in the system

The ***file designator*** is a *source string* whose value is a *file name*. The ***file length*** is a *numeric expression*, which, when evaluated and rounded to an integer, specifies the number of records to be created in the file. File size may vary from a minimum of one record to a maximum of 32,767 records. (This value may be limited by account and system restrictions.) The optional ***record size*** must be a *numeric expression* which, when evaluated and rounded to an integer, gives the file record size in words. The ***record size***, if specified, must be between 64 and 256 words. If not specified, the record size will be set to 256 words. The state of any file created with the CREATE statement is LOCKED.

A LOCKED or PRIVATE program saved in a group library account having the File Create/Purge (FCP) capability which has been executed or chained-to may create a BASIC formatted file in any of the account libraries which are members of that group having the Program/File Accessibility (PFA) capability.

Note that the CREATE statement does not open a file for access. The file must still be opened using a FILES or ASSIGN statement.

Examples:

```
10 CREATE N,"MYFILE",200
20 CREATE M,"HERFIL",500,64
30 CREATE P,A$,100
40 CREATE Q,B$,X**2,J
```

CTL Function

General Form:

CTL (*numeric expression*)

The CTL function can be included in print operations to provide hardware control for ASCII file output devices. For example, PRINT #1; CTL(1) where file number 1 has been equated to a line printer, would cause the line printer to perform a top of form function.

The CTL function can be used in PRINT, PRINT #, PRINT USING, MAT PRINT, MAT PRINT #, and MAT PRINT USING statements. The *numeric expression* is evaluated and rounded to an integer. This value is used to select a device command to be sent to the ASCII device. The CTL function is ignored when directed to BASIC formatted files.

The CTL function is intended for use in ASCII file print operations but can also be used with other print operations by using the *OUT=file name* forms of the RUN and EXECUTE commands.

Any CTL function can be used in print operations directed to any device. When the particular command is not defined for that device the CTL function is ignored. All CTL functions directed at your terminal for example, will be ignored. Defined functions of the CTL function are given in the following table.

Any pending characters generated by preceding print items are output before the CTL command. A comma after a CTL function in a print operation is treated as a semicolon (refer to the discussion of ***print delimiter*** under the PRINT statement).

Examples:

```
10 PRINT #1; A0$(20,50),CTL(X+1),"END OF DATA"
20 PRINT N,M, CTL(3), A$;B$,CTL(3),W(3,3);X(4,4);
```

CTL Function (Cont)

Defined Uses of the CTL Function

VALUE OF CTL ARGUMENT	EFFECT
Line printers use arguments 1 to 13	
1	Print and skip to channel 1 (normally top of form)
2	Print and skip to channel 2 (normally bottom of form)
3	Print and skip to channel 3 (normally next line)
4	Print and skip to channel 4 (normally next double line)[1]
5	Print and skip to channel 5 (normally next triple line)[2]
6	Print and skip to channel 6 (normally next half page)
7	Print and skip to channel 7 (normally next quarter page)
8	Print and skip to channel 8 (normally next sixth page)
9	Print and skip to channel 9 (installation defined)[3]
10	Print and skip to channel 10 (installation defined)[3]
11	Print and skip to channel 11 (installation defined)[3]
12	Print and skip to channel 12 (installation defined)[3]
13	Print and suppress spacing (suppresses paper advance)[4]
Magnetic tapes use arguments 20 to 24	
20	Skip forward to tape mark (the tape is positioned just before the next tape mark)
21	Skip to next file (the tape is positioned just past the next tape mark; the end of file condition occurs if the end-of-tape mark is read before the next end-of-file)
22	Skip backward to tape mark (the tape is positioned just past the preceding tape mark)
23	Skip to preceding file (the tape is positioned just past the second preceding tape mark; the end of file condition occurs if the beginning-of-tape mark is read before the first preceding end-of-file mark)
ASCII disc files use argument 24	
24	Logically rewind file (reposition the file pointer to the beginning of the file)
Paper tape punches use arguments 30 to 33	
30	Output data with even parity and with the X-OFF, RETURN, and LINE FEED characters as a record separator (this is the default mode)[5]
31	Output data with odd parity and with the X-OFF, RETURN, and LINE FEED characters as a record separator[5]
32	Output data with no parity and with the X-OFF, RETURN, and LINE FEED characters as a record separator[5]
33	Output data with no parity and no record separators (this can be used for binary output)[5]

Notes to CTL argument functions:

[1]Skipping to a double space line is not always equivalent to double-spacing, since the current line may not be a double space line itself.

[2]Skipping to a triple space line is not always equivalent to triple-spacing.

[3]Not all line printers have channels 9-12. The action taken for these devices will be as for CTL(3); i.e., single spacing.

[4]Use of this argument prints any pending characters and suppresses spacing; the next line will overprint the current line. Not all line printers have this capability; those that do not will single space.

[5]A 12-inch leader and trailer is automatically punched. Should a program using the tape punch terminate abnormally (error, BREAK key, or disconnect), no trailer will be punched.

DATA Statement

General Form:

DATA *constant list*

The DATA statement specifies data for READ statements.

A ***constant list*** is one or more *constants* separated by commas.

The data is read in sequence from first to last DATA statement, and from left to right within a given DATA statement.

DATA statements may be placed anywhere in a program. The data items will be read in sequence as required by READ statements. The RUN and EXECUTE commands and a successful CHAIN statement reset the data pointer to the first item in the first DATA statement of the program.

The TYP function may be used to test the type of data items and the RESTORE statement may be used to move the data pointer without performing a read operation.

Examples:

```
10 DATA 457,"STRING DATA",2.192
20 DATA "AB" '65 '66 '7'10'13
```

See also:

READ
TYP
RESTORE

DEF Statement

General Form:

DEF *function name(parameter)* = *numeric expression*

The DEF statement allows you to define special functions within a program.

A user defined function is one that is defined within the user program and is called within that program in the same way that one of the system-defined functions (i.e., SIN, SQR, TAN, etc.) is called.

The ***function name*** is made up of the letters FN followed by one of the letters A through Z (for example, FNC). The ***parameter*** has the form of a *numeric simple variable*. The *numeric simple variable* is a "dummy" variable whose purpose is to indicate where the actual argument of the function will be used when called. For example, in the following sequence, M is a dummy variable:

```
10 LET Y = 100
20 DEF FNA(M) = M/10
30 PRINT FNA(Y)
40 END
RUN
 10
```

When FNA(Y) is called for in statement 30, the formula defined for FNA in statement 20 is used to determine the value printed.

Any operand in the program may be used in the defining expression; however, circular definitions such as:

```
10 DEF FNA(Y) = FNB(X)
20 DEF FNB(X) = FNA(Y)
```

cause infinite looping.

Examples:

```
60 DEF FNA(B2) = A**2 + (B2/C)
70 DEF FNB(B3) = 7*B3**2
80 DEF FNZ(X) = X/5
```

DIM Statement

General Form:

DIM *dimension list*

The DIM statement sets the amount of space allocated by the system for arrays and strings.

The ***dimension list*** is one or more dimension elements, separated by commas. A dimension element is a *string simple variable* followed by the string length in parentheses, an *array name* followed by the number of rows (one dimension array) in parentheses, or an *array name* followed by the number of rows and columns (two dimensional array) in parentheses.

DIM statements may occur at any point in the program. A variable may appear only once in DIM statements in a given program. The COM statement may be used instead of the DIM statement.

Example:

```
20 DIM A(5),B(12,7),G$(240),C(5,5),R0$(90)
```

END Statement

General Form:

END

The END statement terminates execution of the program and returns control to the system.

The END statement may occur at any point in the program and must be used as the last statement in all programs.

Example:

```
1000 END
```

ENTER Statement

General Form:

ENTER ***#numeric variable***

or

ENTER [***#numeric variable,***] ***time allowed,return variable,read variable***

The ENTER statement allows you to have greater control over data input than that available with the INPUT and LINPUT statements.

The ENTER statement may be used to limit the time allowed for data input, to test the actual response time of a given input, to determine the user's port number, and to input data for one ***read variable***.

If the number sign (#) and *numeric variable* are provided, the system stores the user's port number in the *numeric variable* as a value in the range 0 to 31.

The ***time allowed*** is a *numeric expression* which, when evaluated and rounded to an integer, specifies the time allowed for response in seconds. This expression should evaluate to a number (1 to 255). Zero is treated as one; numbers less than zero or greater than 255 are treated modulo 255 (256 = 0, 257 = 1, etc). Timing begins when all previous statements have been executed and any resultant output to the user terminal has been printed.

The *return variable* will indicate the precise time in seconds the user took to respond. If the response is not acceptable, such as wrong data type, the value is the negative of the response time and the read variable remains unchanged. If the user failed to respond within the time limit, the value is set to −256. If a parity error occurred in transmission, −257 is returned. If a character was lost in transmission, −258 is returned.

The ***read variable*** is a *destination string* or a *numeric variable*. A character string being entered need not be enclosed in quotes, but may contain quotes and blanks. The extended literal string form ('71, '23, etc.) is not recognized. A carriage return alone is interpreted as a null string. If the *destination string* is not doubly subscripted and the character string entered is too long to fit in the *physical length* of the *string*, the *return variable* is negated (refer to the description of *destination string*). A *numeric variable* must be satisfied with a *numeric constant*.

The ENTER statement differs from the INPUT statement in that a "?" prompt is not displayed on the user terminal following data input and a linefeed is not generated following execution of the ENTER statement. The next program statement is executed whether or not the input time limit is exceeded.

Examples:

```
100 ENTER #V              (your port # is returned in V)
200 ENTER A,B,C$          (you have A seconds to input C$; B is the actual time taken)
300 ENTER #V,K1,K2,K3     (you have K1 seconds to input K3; V is set to your port # and
                           K2 is the actual time taken)
400 ENTER 25,L,Q          (you have 25 seconds to input Q; L is the actual time taken)
```

EXP Function

General Form:

EXP (*numeric expression*)

The EXP function is a numeric-valued function which returns the mathematical constant "e" raised to the power of the *numeric expression* ($e^{\text{numeric expression}}$).

The approximate value of the constant e is 2.718282.

Example:

```
50 A = B*EXP(M/10)
```

FILES Statement

General Form:

FILES *file list*

The FILES statement opens files for use in a program.

The FILES statement does not create files (see the CREATE and FILE commands and the CREATE statement). The ***file list*** is made up of *file names* separated by commas. Up to four FILES statements can appear in a program, but only 16 files total can be declared (duplicate entries are allowed). The files are numbered (from 1 to 16) in the order they are declared in the program. These *file numbers* are used by READ, PRINT, LINPUT, ADVANCE, UPDATE, ASSIGN, and IF END statements, and TYP, REC, and ITM functions for file access and control.

A "*" may be used in the file list to reserve a position for a file to be named at a later point in the program using the ASSIGN statement. For example, 10 FILES AFILE, *, BFILE reserves file position number 2 for a file to be assigned later in the program. Note that a file name must be assigned to the reserved file number before a read, write, or test using the file number occurs, otherwise an error will result.

System library files may be accessed by using a "$" in front of the file name ($FRED). Group library files may be accessed by using a "*" in front of the file name (*TFILE). Files in other accounts may be accessed by adding the owner's idcode to the file name (file name.idcode) (HIS FIL.C901). Access to any files outside your own library are subject to that account's file access capabilities and restrictions.

If a file cannot be opened (the file does not exist or you are not allowed access), the program will terminate. If you are granted read-only access to the file because of its status or because it is in use, you will not be notified of this condition until you attempt to PRINT, which will cause program termination. Use of the ASSIGN statement (instead of FILES) to open the file will allow test of a *return variable* value to determine read/write access.

FILES Statement (Cont)

The system uses part of the user workspace for control information concerning the file approximately 18 words per *file name* position mentioned in the FILES statement. In addition, an area equal to the record length (1 to 1024 words) of the file is used in the user workspace as a buffer area for each file that is opened. A "*" used to reserve a position for a file in the FILES statement uses 256 words as a potential buffer. An ASCII disc file always uses 256 words regardless of the record length specification.

The same *file name* may be mentioned more than once in FILES statements used in a program to take advantage of the in-memory buffering of the file data described above (10 FILES AFILE,*,AFILE). During processing of the FILES statement, the checking the system normally performs to determine whether a file being opened is already in use is suspended (only for BASIC formatted files) for each occurrence of the same *file name* after the first reference to that *file name*. (The checking is not suspended if the file is open in another program.)

Files are opened when program execution begins, not when the FILES statement is encountered by the executing program. Thus, all FILES statements should refer to previously created files. If you desire to open a file created programatically, use the ASSIGN statement.

Examples:

```
200 FILES MYFILE,$SYSFL,*GPFL,*,YOURFL.M350
300 FILES KEN,JIM,KEN,*,JIM
```

FOR and NEXT Statements

General Form:

FOR *for variable* = *initial value* TO *final value* [**STEP *step size***]
NEXT *for variable*

The looping statements FOR and NEXT allow you to repeat a group of statements a specified number of times.

The FOR statement precedes the statements to be repeated, and the NEXT statement directly follows them. The number of times the statements are repeated is determined by the value of the ***for variable***, which is a *numeric simple variable*. The ***initial value***, ***final value***, and the ***step size*** all are *numeric expressions*.

When the FOR statement is executed, the ***for variable*** is set to the ***initial value***. Then the following steps occur:

1. The value of the ***for variable*** is compared to the ***final value***; if the ***for variable*** exceeds the ***final value*** (or if the *for variable* is less than the ***final value*** if the ***step size*** is negative), control skips to the statement following NEXT.

2. Statements between the FOR statement and the NEXT statement are executed in normal sequence order.

3. The ***step size*** (or 1 if the ***step size*** was not specified) is added to the value of the ***for variable***.

4. Return to step 1.

Note that round-off errors may increase or decrease the number of steps (times through the loop) when non-integer step sizes are used. The user should not execute the statements in a FOR loop except through a FOR statement. Transferring control into the middle of a loop can produce undesirable results.

Examples:

```
100 FOR P=1 TO 5
      .
      .
      .
170 NEXT P
200 FOR R2=N TO X STEP -1.5          (where N≥X at start)
      .
      .
      .
220 NEXT R2
250 FOR S=1 TO (Z*B) STEP (Y**2-V)
      .
      .
      .
260 NEXT S
```

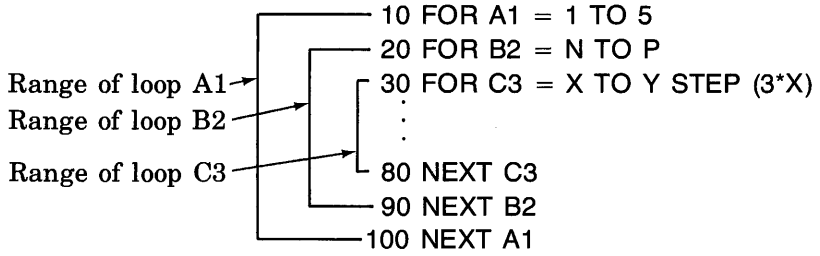
FOR and NEXT Statements (Cont)

Sample Program with a variable number of loops:

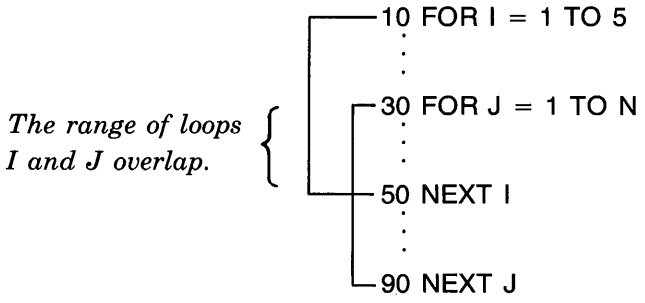
```
 40 PRINT "HOW MANY TIMES DO YOU WANT TO LOOP";
 50 INPUT A          (where A≤10)
 60 FOR J = 1 TO A
 70 PRINT "THIS IS LOOP";J
 80 READ N1,N2,N3
 90 PRINT "THESE DATA ITEMS WERE READ:"N1;N2;N3
100 PRINT "SUM = " ;N1+N2+N3
110 NEXT J
120 DATA 5,6,7,8,9,10,11,12
130 DATA 13,14,15,16,17,18,19,20,21
140 DATA 22,23,24,25,26,27,28,29,30
150 DATA 31,32,33,34
160 END
```

NESTING FOR . . . NEXT LOOPS. FOR . . . NEXT loops may be nested (placed inside one another) as long as the loops do not overlap.

Proper Nesting



Improper Nesting



GOSUB and RETURN Statements

General Form:

GOSUB *statement number*

or

GOSUB *numeric expression* OF *statement number list*
RETURN

The GOSUB statement overrides the normal sequential order of statement execution by transferring program control to the beginning of a subroutine.

A subroutine consists of a collection of statements that may be executed from more than one location in a program. In a subroutine, there is no explicit indication in the program as to which statements constitute the subroutine. The *statement number* to which control is transferred must be an existing statement in the current program. If the GOSUB transfers control to a statement that cannot be executed (such as REM, DIM, COM, DEF), control passes to the next sequential statement after the non-executable statement. A RETURN statement in the subroutine returns control to the statement following the GOSUB statement. There may be more than one RETURN statement in a subroutine.

Example:

```
 50 READ A2
 60 IF A2 > 100 THEN 80
 70 GOSUB 400
  .
  .
  .
380 STOP          (STOP frequently precedes the first statement of a subroutine to pre-
                  vent accidental entry)
390 REM THIS SUBROUTINE ASKS FOR A 1 OR 0 REPLY
400 PRINT "A2 IS <= 100"
410 PRINT "DO YOU WANT TO CONTINUE?";
420 INPUT N
430 IF N # 0 THEN 450
440 LET A2 = 0
450 RETURN
  .
  .
  .
600 END
```

GOSUB and RETURN Statements (Cont)

MULTIBRANCHING. Multibranch GOSUB statements use the value of a *numeric expression* to select the destination statements. The *numeric expression* is evaluated and rounded to an integer "n". Control is then transferred to the "nth" statement in the ***statement number list***. The ***statement number list*** is one or more *statement numbers* separated by commas. Values of n less than 1 or greater than the number of branch statements in the ***statement number list*** are ignored.

Examples:

```
20 GOSUB 3 OF 100,200,300,400,500
60 GOSUB G-1 OF 200,210,220
70 GOSUB R OF 80,180,280,380,480,580
```

NESTING GOSUB STATEMENTS. Another subroutine can be called from within a subroutine. This is known as ***nesting***. GOSUB statements may be nested logically to a level of 20. Nesting more than 20 GOSUB statements without an intervening RETURN statement will cause an error message. Note, however, that nested subroutines are exited in the reverse of the order in which they were entered (last in — first out). For example, if subroutine 250 (below) is entered from subroutine 200, 250 will be exited before subroutine 200.

Examples:

```
100 GOSUB 200
 .
 .
200 LET A = R2/7
210 IF A THEN 230
220 GOSUB 250
 .
 .
250 IF A > B THEN 270
260 RETURN
270 GOSUB 600
```

GO TO Statement

General Form:

GO TO *statement number*

or

GO TO *numeric expression* OF *statement number list*

The GO TO statement overrides the normal sequential order of statement execution by transferring program control to the specified *statement number*.

The *statement number* to which control is transferred must be an existing statement in the current program. If the GO TO statement transfers control to a statement that cannot be executed (such as REM, DIM, COM, DEF), control passes to the next sequential statement after the non-executable statement.

When the second form of the GO TO statement is used, the *numeric expression* is evaluated and rounded to an integer "n". Control then is transferred to the "nth" *statement number* in the ***statement number list***, where ***statement number list*** is one or more *statement numbers* separated by commas. If there is no *statement number* corresponding to the value of the *numeric expression*, the GO TO statement is ignored and the statement following the GO TO statement is executed.

Extreme caution should be used if a GO TO statement is used to enter a FOR . . . NEXT loop. Doing so may produce an unpredictable result.

Examples:

```
10 GO TO X - Y OF 400,500,600
50 GOTO 100
80 GOTO 10
90 GOTO N OF 100,200,300
```

IDN Function

(refer to MAT)

IF . . . THEN Statement

General Form:

IF *numeric relation* THEN *statement number*

or

IF *string relation* THEN *statement number*

The IF . . . THEN statement transfers control to the specified *statement number* if the ***numeric relation*** or ***string relation*** is true. If the specified condition for transfer is not true, the statement following the IF . . . THEN statement will be executed.

A ***numeric relation*** is a *numeric expression*. The *numeric expression* is evaluated to a number which is considered "true" if the value is non-zero, and "false" if the value is zero. Because numeric values may be rounded during computation, the "=" operator should be used carefully in IF . . . THEN statements. It is recommended that the "<=" or ">=" operators be used instead of "=" whenever practical when arithmetic computation is involved.

Example:

```
IF A < = B AND Z/2 > = M THEN 120
```

A ***string relation*** is composed of a *string variable* followed by a *relational operator* followed by a *string expression* (for example, A$ = "ABC"). Strings are compared character by character within the two strings on either side of the *relational operator*. A given character is "less than" another character if the given character occurs in a lower position in the ASCII collating sequence (Appendix A) than the other character.

A character is "greater" if it occurs in a higher position in the ASCII collating sequence. A character is equal to another character only if they are the same character. (For example, "A" < "B" and "Z" > "1".) Note that all characters have a value (a position in the ASCII collating sequence) including blanks and non-printing characters.

If strings of unequal lengths are compared, and the characters in the shorter string are identical with the initial characters in the longer string, the shorter string is "less than" the longer string.

Two strings are "equal" only if they contain identical characters and are both of the same length.

Examples:

```
10 LET N=10
20 READ X
30 IF X<N THEN 60
40 PRINT "X IS 10 OR OVER"
50 GOTO 80
60 PRINT "X IS LESS THAN 10"
70 GOTO 30
80 END
```

```
10 INPUT A$
20 IF A$="DONE" THEN 50
30 PRINT#1;A$
40 GOTO 10
50 END
```

```
10 READ#1,R;V
20 IF V>(R**2-1) OR R=3 THEN 50
30 V=V**2
40 GOTO 10
50 END
```

IF END Statement

General Form:

IF END # *file number* THEN *statement number*

The IF END statement causes a branch to a specified statement when an End-of-File (EOF) mark is detected during a file read operation; an attempt is made to write beyond the end of a file in a write operation; or a direct file write exceeds the record.

Since the IF END statement only sets a flag which stays set until another IF END statement is executed for the same file, or until the file is closed, it is not necessary to execute the statement prior to every file read or write.

Note that the branch is not taken if an EOF is encountered during execution of an ADVANCE statement.

If the IF END statement is not used and an EOF condition occurs during a file read or write operation, an error will occur terminating the program.

The statement will remain in effect throughout the program until changed by another IF END statement referencing the same file number. Note that the statement is associated with a *file number* and not a *file name*. The IF END trap is disarmed if the file is de-assigned.

The following example reads strings from a file until an EOF is reached, at which point the program informs the user that an EOF was encountered.

Example:

```
10 FILES FIL1
20 IF END #1 THEN 70
40 READ #1;A$
50 PRINT A$
60 GOTO 40
70 PRINT "END OF FILE"
80 END
```

IMAGE Statement

General Form:

IMAGE *format string*

The IMAGE statement is used to represent the ***format string*** for a PRINT USING or MAT PRINT USING statement. See the PRINT USING statement discussion for a description of the contents of a ***format string.*** Note that a ***format string*** in an IMAGE statement is *not* enclosed in quotes.

Example:

```
10 IMAGE # , 2(3DX/),"BOW-WOW",3AX
```

INPUT Statement

General Form:

INPUT *read variable list*

The INPUT statement allows you to input data from your terminal during program execution.

When the INPUT statement is executed, a "?" is displayed on the terminal and the program pauses until the input requirements are satisfied.

A ***read variable list*** is one or more ***read variables*** separated by commas. A ***read variable*** is a *numeric variable* or a *destination string*. For example, N, Y(3), A$, and B0$(6,7) are all ***read variables***.

To respond to the "?" request for INPUT, you must type a list of *numeric constants* and/or *strings* separated by commas. The input data must be of the same type as the variables in the ***read variable list***. If the data type does not agree with the variable type, you will receive the message "BAD INPUT, RETYPE FROM ITEM n," where n is the item number that was mistyped. Note that this item number refers to the line just typed. It does not refer to previous lines entered already in response to the same INPUT statement. If insufficient items are input, a "??" will be displayed on your terminal. If more items are input than were requested, you will receive the message "EXTRA INPUT — WARNING ONLY" and the additional items will be discarded. Should the system not properly receive the line typed, you will receive the message "TRANSMISSION ERROR, REENTER."

When entering a *numeric constant*, all characters before the first "+", "−", ".", or digit are ignored. Otherwise, the form of the number corresponds to the definition given under BASIC language elements.

When entering a *string*, leading blanks are ignored unless an opening quote is included. Trailing blanks are not ignored and a closing quote must be included unless the string is the last item entered on a line (or only item entered).

The only way to stop a program when input is required is to press the BREAK key. This assumes that the break function has not been disabled (refer to the BRK function). The BREAK key terminates the program. The program must be restarted using the RUN or EXECUTE command.

Examples:

```
10 INPUT A
20 INPUT A$,B,C(3)
30 INPUT P1$(N,N+20)
```

INT Function

General Form:

INT (*numeric expression*)

The INT function is a numeric-valued function which returns the integer part of the *numeric expression*. The integer part of a number is that integer less than or equal to the number. For example, INT(3.5) = 3 but INT(−3.5) = −4.

Examples:

```
10 LET A = INT(X**Y)
20 PRINT INT(Z)
```

INV Function

(Refer to MAT . . . INV)

ITM Function

General Form:

ITM (*numeric expression*)

The ITM function returns the number of data items (numbers and strings) between the beginning of the currently accessed file record and the position of the file pointer for a file.

The *numeric expression*, when evaluated and rounded to an integer, is used as the *file number* to refer to the file (refer to the discussion of files in Section V).

ITM cannot be used with ASCII files; if attempted the program will be terminated with an error. Specification of an invalid *file number* will result in an error.

Example:

```
20 PRINT "I WILL NOW READ ITEM" ITM(6) "FROM FILE"6
```

LEN Function

General Form:

LEN (*source string*)

The LEN function returns the current (logical) length in characters for the specified string.

The DIM statement specifies a maximum (physical) string length. The LEN function, however, allows you to check the actual number of characters currently assigned to a string variable. Do not confuse the LEN function with the LENGTH command.

Examples:

```
100 A = LEN(B$)
200 X$(LEN(X$)+1) = "ADDITIONAL SUBSTRING"
300 IF LEN(A$)#3 THEN 400
400 GOTO LEN(G$) OF 500,600,700,800
```

LET Statement

General Form:

[LET]*replacement list* = *numeric expression*

or

[LET]*destination string* = *string expression*

The LET statement assigns a value to one or more variables (10 LET B = 16.3).

The value assigned by the LET statement may be in the form of a *numeric expression* or a *string expression*. The ***replacement list*** is one or more *numeric variables* separated by replacement (assignment) operators ("="). In the LET statement, the equal sign ("=") is an assignment operator. It does not indicate equality, but is a signal that the value on the right of the assignment operator is to be assigned to the variable(s) on the left.

If a value is assigned to more than one variable, the assignment is made from right to left. For example, in the statement A = B = C = 2, first C is assigned the value 2, then B is assigned the current value of C, and, finally, A is assigned the value of B. When a subscripted variable is to be assigned a value, its subscript is evaluated first and then the expression. Thus A(I) = I = (T+R)/S is equivalent to I = A(I) = (T+R)/S.

The rule that the equal sign ("=") is an assignment operator only holds as long as *numeric variables* occur in the ***replacement list***. If a *numeric constant* appears, followed by an equal sign, that equal sign is treated as a relational operator. For example, in A=2=B the first "equal sign" (=) is treated as an assignment operator and the second is treated as a relational operator.

Note that the word LET is an optional part of the assignment statement.

Examples:

```
10 LET A = 5.02
20 A = 5.02
30 X = Y7 = Z = Z(X) = 1
40 B$ = "ABC"
50 B4(1,5) = "ABCDE"
60 Z1$ = A1$(3,5)
70 M$(N,M) = A$(N,M)
```

LIN Function

General Form:

LIN (*numeric expression*)

The LIN function can be included in print operations to perform a carriage return and one or more line feed operations.

The LIN function can be used in PRINT, PRINT #, PRINT USING, MAT PRINT, MAT PRINT #, and MAT PRINT USING statements. The *numeric expression* is evaluated and rounded to an integer. If the value is positive, the value of the expression specifies the number of line feeds to be generated. If the value is negative, the absolute value of the *numeric expression* will be used to determine the number of line feeds and the initial carriage return is suppressed.

Note that unless there is a trailing comma or semicolon at the end of the PRINT statement, the normal X-OFF, carriage return, and line feed characters will be generated in addition to those generated by the LIN function.

The LIN function is ignored in file print (PRINT # and MAT PRINT #) operations to BASIC formatted files but is allowed in print operations to ASCII files.

A comma following a LIN function in a print statement is treated as a semicolon (refer to the discussion of ***print delimiter*** under the PRINT statement).
Examples:

```
27 PRINT A, LIN(M*N/C1)
28 MAT PRINT C;LIN(10),D
29 PRINT M3,M(4,5),LIN(-A)
```

```
10 PRINT "ABC";LIN(-1);"DEF";LIN(2);"GHI"
20 END
RUN


ABC
   DEF

GHI

DONE
```

```
10 PRINT TAB(8);" TITLE:PRINT HEADING";SPA(10);"SUMMARY REPORT";
20 PRINT LIN(3);" DETAIL LINES"
30 END
RUN


        TITLE:PRINT HEADING          SUMMARY REPORT


 DETAIL LINES

DONE
```

LINPUT Statement

General Form:

LINPUT *destination string*

The LINPUT statement accepts an entire line of string data from the user terminal and assigns it to a *destination string*.

No prompt character is printed when the LINPUT statement is used.

All characters entered (including commas, quote marks, and leading and trailing blanks) are assigned to the string.

Examples:

```
30 LINPUT A$
60 LINPUT X1$(10,20)
```

LINPUT # Statement

General Form:

LINPUT *#file number;destination string*

The LINPUT # statement reads the contents of the next available record into a *destination string*. The referenced file must be an ASCII file.

Successive reads of the ASCII file cause successive records to be accessed. It is not possible to read the remainder of a record partially read by a preceding READ# statement.

Examples:

```
510 LINPUT #5;A$
520 LINPUT #1;B$(1,80)
```

LOCK Statement

General Form:

LOCK # *file number* [, *return variable*]

The LOCK statement is used to set or test a file status flag.

A file status flag is associated with each BASIC formatted file (refer to FILES Section V). The LOCK statement may be used to set the flag for a specific file (using the *file number*), indicating that you want exclusive access to the file. By using a *return variable* and testing its value, you can determine if the file is already locked. Similarly, once successfully "locked" by your program, other programs which check the status of the file will be alerted to the fact that the file is locked.

If no *return variable* is supplied, a program may only have one file locked at any given time. If a *return variable* is supplied, more than one file may be concurrently locked. If no *return variable* is supplied and the file is currently locked by some other program, your program will not resume execution until the previous program unlocks the file.

The file status flag is cleared by the UNLOCK statement, chaining to another program, or program termination. All pending write operations (from the locking program) are completed before the flag is cleared. The LOCK statement should always be used with a matching UNLOCK statement.

The meaning of the *return variable* for LOCK is as follows:

Return Value	Meaning
0	File locked successfully
1	File already locked (by your program or another program)
2	Invalid file number

Since the LOCK statement does not deny others access to the locked file, users must cooperate in its use for it to be effective. Locking is not necessary if more than one person does not access the file at the same time or if the file is not written to.

Examples:

```
 20 FILE AFILE,BFILE
 30 LOCK #1
 40 READ #1;N
 50 UNLOCK #1
 60 LOCK #2,R
 70 GOTO R + 1 OF 80,90,100
 80 PRINT #2;N
 85 UNLOCK #2
 86 GOTO 30
 90 PRINT "FILE BUSY"
 95 GOTO 110
100 PRINT "FILE NUMBER ERROR"
110 STOP
```

LOG Function

General Form:

LOG (*numeric expression*)

LOG is a numeric valued function that returns the natural logarithm, i.e., log to the base "e" ($\log_e$(*numeric expression*)).

The *numeric expression* must evaluate to greater than zero. If the evaluation results in zero, a warning message will be produced and the value of the log function is set to -10^{38}. If the argument is evaluated to a negative value (less than zero), the program terminates with an error.

Examples:

```
10 PRINT LOG(X)
20 A = LOG(B)
```

MAT Addition and Subtraction Statements

General Form:

MAT *array name* = *array name* + *array name*

or

MAT *array name* = *array name* − *array name*

The MAT addition statement sets an array equal to the sum of two arrays of the same dimensions. Addition is element by element (A(I,J) = B(I,J) + C(I,J)). The MAT subtraction statement sets an array equal to the difference of two arrays of the same dimensions. Subtraction is element by element (A(I,J) = B(I,J) − C(I,J)).

The dimensions of the resultant array must be the same as the component arrays. The same array may appear on both sides of the "=" sign.

Examples:

```
100 DIM A(20), B(20), C(20)
200 MAT A = B + C
300 MAT A = A + B
350 MAT A = A + A
400 MAT A = B - C
500 MAT A = A - B
```

MAT Assignment Statement

General Form:

MAT *array name* = *array name*

The MAT assignment statement is used to set one array equal to another array of the same dimensions. The transfer is element by element (A(I,J) = B(I,J)).

Example:

```
10 DIM A(10,20), B(10,20)
20 MAT B = A
```

MAT. . .CON Statement

General Form:

MAT *array name* = CON [*new dimensions*]

The MAT. . .CON statement sets up an array with all elements equal to one. The *new dimensions* parameter is optional.

Examples:

```
205 MAT C = CON
210 MAT A = CON(N,N)
220 MAT Z = CON(5,20)
230 MAT Y = CON(50)
```

MAT. . .IDN Statement

General Form:

MAT *array name* = IDN [*new dimensions*]

The MAT. . .IDN statement is used to establish an identity matrix (all 0's with a diagonal of all 1's).

Note that the array must contain the same number of rows and columns (must be square), or the optional *new dimensions* parameter must re-specify the dimensions as equal.

Sample identity matrix:

```
1   0   0
0   1   0
0   0   1
```

Examples:

```
205 MAT A = IDN
210 MAT B = IDN(3,3)
215 MAT Z = IDN(Q5,Q5)
220 MAT S = IDN(6,6)
```

MAT INPUT Statement

General Form:

MAT INPUT *array read list*

The MAT INPUT statement allows you to input entire arrays from the terminal.

An ***array read list*** is one or more *array names*, separated by commas. Each *array name* may optionally be followed by a *new dimensions* specification. The prompt characters and error messages used for the INPUT statement are used for the MAT INPUT statement.

Elements entered must be *numeric constants* (see the INPUT statement for special rules on entering *numeric constants*). Elements must be separated by commas and entered row by row, i.e., all of the elements in row 1 are entered before the elements in row 2, etc.

Note the difference between the MAT INPUT statement and the INPUT statement. For example, INPUT X(3,5) causes the fifth element of the third row to be input, while MAT INPUT X(3,5) causes the entire array X to be input, and sets its working size to 3 x 5.

Examples:

```
27 MAT INPUT A,B
30 MAT INPUT X,Z(15,N)
```

MAT. . .INV Statement

General Form:

MAT *array name* = INV(*array name*)

The MAT. . .INV statement is used to invert an array.

If array A is square and non-singular (determinant of A $<>$ 0), there exists a unique array (invert of A) such that A times the invert of A yields the identity matrix (see the MAT. . .IDN statement). For additional information on array inversion refer to the discussion of ARRAYS in Section III.

If you attempt to invert a singular array (determinant = 0), your program will terminate with an error.

Both arrays used in a MAT. . .INV statement must be square and be of the same dimensions. The same array may be used on both sides of the equation.

In performing the inversion, the system uses a temporary array requiring storage equal to the array being inverted. In some cases this may limit the size array that may be inverted.

Examples:

```
200 MAT A = INV(A)
300 MAT B = INV(A)
```

MAT Multiplication Statement

General Form:

MAT *array name* = *array name 1* * *array name 2*

Array multiplication sets an array equal to the product of two arrays.

The array which contains the final value must be large enough to hold the number of elements resulting from multiplying the number of rows in *array name 1* by the number of columns in *array name 2*. The number of columns in *array name 1* must be equal to the number of rows in *array name 2*. If array 1 is A(P,N) and array 2 is B(N,Q) then the resulting array would be C(P,Q) and would contain P × Q elements. For example, if A is multiplied by B:

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

where the dimensions are A(3,2) and B(2,4), then C will have dimensions of (3,4). The same variable cannot appear on both sides of the equation.

Example:

```
200 MAT C = A * B
```

MAT PRINT Statement

General Form:

MAT PRINT *array print list* [***print delimiter***]

The MAT PRINT statement is used to print entire arrays in a single statement.

The ***array print list*** is any combination of *array names* and ***print functions*** separated by commas or semicolons (which are ***print delimiters***). The ***array print list*** may be followed by an optional ***print delimiter*** (comma or semicolon), which controls output formatting.

A ***print function*** is one of the following:

- TAB (*numeric expression*) — tabs to column position
- LIN (*numeric expression*) — generates line feeds
- SPA (*numeric expression*) — spaces
- CTL (*numeric expression*) — outputs special device control commands

The elements of the array are printed row by row and can be spaced out or packed together (using commas or semicolons), as in the PRINT statement.

Each row of each array is printed separately, with double spacing between rows. If a comma follows the array, each element starts in one of the five divisions of the line. The output line is divided into five consecutive fields: four fields of 15 characters each and one field of 12 characters (starting at columns 0, 15, 30, 45, and 60) for a total of 72 characters. If a semicolon follows the array, the elements are printed packed together, as if each element were followed by a semicolon. If nothing follows the last array, a comma is assumed. All formatting is done according to the specifications under PRINT statement.

A one-dimensional array is printed as a single column.

Note that individual elements of an array can be printed using PRINT. For example,

```
100 PRINT A(1),A(2),C(50)
```

Examples:

Note the effect of the semicolons following A and B in the MAT PRINT statement, line 70, on the printed output. MAT READ in line 20 redimensions array B; redimensioning of arrays is not permitted in a MAT PRINT statement.

```
10  DIM A[10], B[5,5], C[2,2]
20  MAT  READ A, B[3,5], C
30  MAT  PRINT A
40  PRINT
50  MAT  PRINT B
60  PRINT
70  MAT  PRINT A; B;
80  DATA 2.5, 46.7, 75, 0, 50.1, 0, 0, 0, 19.8, 0
90  DATA 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
100  DATA 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
110  END
```

MAT PRINT Statement (Cont)

```
RUN

 2.5
 46.7
 75
 Ø
 5Ø.1
 Ø
 Ø
 Ø
 19.8
 Ø

 1             2             3             4             5
 6             7             8             9             1Ø
 11            12            13            14            15
 2.5
 46.7
 75
 Ø
 5Ø.1
 Ø
 Ø
 Ø
 19.8
 Ø
 1       2       3       4       5
 6       7       8       9       1Ø
 11      12      13      14      15

DONE
```

MAT PRINT # Statement

General Form:

MAT PRINT # *file number* [, *record number*] ; END

or

MAT PRINT # *file number* [, *record number*] ; *array write list* [*print delimiter*]

The MAT PRINT # statement prints arrays specified in the ***array write list*** on the specified file row by row. END which can be the last or only item in the *array write list* writes an end-of-file mark on the file.

A serial print (no *record number* specified) prints an entire *array* on the file starting at the current position of the file pointer. A direct print prints the entire array starting at the beginning of the record specified in *record number*. An *array write list* is any combination of *array names* and *print functions* separated by ***print delimiters***. In addition, the last or only item in the *array write list* can be END. A ***print delimiter*** is either a comma or a semicolon. Refer to PRINT # for a discussion of writing to files. Note that MAT PRINT # writes an entire array where PRINT # writes single elements.

Example:

The following example writes two arrays onto file BB. It then reads the array values into two different arrays with different dimensions.

```
10  FILES BB
20  DIM A[2,5],B[3,2]
30  MAT  READ A
40  MAT  READ B
50  DATA 10,20,30,40,50,60,70,80,90,100,1,2,3,4,5,6
60  MAT  PRINT #1;A,B
70  DIM C[5,2],D[2,3]
80  MAT  READ #1,1;C,D
90  MAT  PRINT C,D
100  END
RUN


 10             20

 30             40

 50             60

 70             80

 90             100

 1              2              3

 4              5              6

DONE
```

MAT PRINT USING Statement

General Form:

MAT PRINT USING ***format part*** [**;** ***array print list***]

The MAT PRINT USING statement allows you to control the output format of data from arrays. The ***format part*** is either a ***format string*** (represented as a *literal string* or *string variable*) or a *statement number* referencing an IMAGE statement. Refer to the PRINT USING statement for a description of the contents of a *format string*, noting that *string format specifications* are not allowed with MAT PRINT USING statements.

The optional ***array print list*** is any combination of *array names* and ***print functions*** (TAB, LIN, SPA, or CTL) separated by commas. Semicolons and trailing punctuation are not allowed. The commas serve as delimiters only; they have no formatting function. As with the MAT PRINT statement, the matrices are printed in row order.

Examples:

```
 9 IMAGE 10(DX)/
10 MAT PRINT USING 9; A, LIN(2)
20 MAT PRINT USING "S6D.2DXE/"; B,SPA(3),C
30 A$="3D.D"
40 MAT PRINT USING A$; A,B,C
50 A$='34"RIDICULOUS EXAMPLE" '34
60 MAT PRINT USING A$
```

MAT READ Statement

General Form:

MAT READ *array read list*

The MAT READ statement reads entire arrays from DATA statements.

An ***array read list*** is one or more *array names* separated by commas. Each *array name* may optionally be followed by a *new dimensions* specification.

Example:

```
10 MAT READ A,B(10,12)
```

See also:

DATA

MAT READ # Statement

General Form:

MAT READ # *file number* [, *record number*] ;*array read list*

The MAT READ # statement reads array data from a file.

If the optional *record number* is provided, reading begins with the data in that record. An ***array read list*** is one or more *array names*, separated by commas. Each *array name* may optionally be followed by a *new dimensions* specification.

When a file is accessed by record and more items are requested than exist in the record, an end-of-file (EOF) condition is generated. Serial file reads are limited only by the length of the file. An ASCII file may be referenced, but data values are read in the same manner as in the INPUT statement.

Examples:

```
10 MAT READ #3;B
20 MAT READ #1,3;A(5,6)
30 MAT READ #5,N;A,B,M,Z
```

MAT Scalar Multiplication Statement

General Form:

MAT *array name* = (*numeric expression*) * *array name*

Scalar multiplication sets an array equal to the product of a number and an array.

Multiplication is element by element (A(I,J) = *numeric expression* * B(I,J)). Both arrays must have the same working size. The same array may appear on both sides of the "=" sign.

Examples:

```
100 MAT A = (5) * B
200 MAT A = (N/3) * A
300 MAT A = (Q7 * N5) * C
```

MAT. . .TRN Statement

General Form:

MAT *array name* = TRN(*array name*)

The MAT. . .TRN statement is used to set an array to the transposition of a specified array.

Transposition causes rows and columns to be switched (row 1 becomes column 1, etc). The same array cannot be used on both sides of the "=" sign.

Sample transposition:

Original Array

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

Transposed Array

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

Note that the dimensions of the resulting array must be the reverse of the original array. For example, if A has dimensions of 6,5, and MAT C = TRN(A), C must have dimensions of 5,6.

Example:

```
300 MAT A = TRN(B)
```

MAT. . .ZER Statement

General Form:

MAT ***array name*** **= ZER** [***new dimensions***]

The MAT. . .ZER statement sets all elements of the specified array to zero. The *new dimensions* parameter is optional.

Examples:

```
305 MAT A = ZER
310 MAT Z = ZER(N)
315 MAT X = ZER(30,10)
320 MAT R = ZER(N,P)
```

NEXT Statement

(Refer to FOR)

NUM Function

General Form:

NUM(*source string*)

The NUM function returns the numeric value (ASCII code value) of the first character in the specified *source string*. For example, NUM("A") would return 65. NUM of a zero length string returns 0.

A list of ASCII character values is given in Appendix A.

Examples:

```
200 PRINT NUM(A$(J,J))
300 GOTO NUM(X$(LEN(X$))) - 60 OF 400,500,600
```

See also:

CHR$

POS Function

General Form:

POS(*source string 1, source string 2*)

The POS function returns the character position in *source string 1* where the first incidence of *source string 2* occurs. For example, POS("12ABC34","ABC") would return 3.

If *source string 2* is not a substring of *source string 1*, then the value returned is 0. If *source string 2* is a null string, then the value returned is 1.

Examples:

```
200 PRINT POS(A$,B$(3,7))
300 A$(J) = B$(POS(B$,G$),POS(B$,G$) + LEN(G$))
```

PRINT Statement

General Form:

PRINT

or

PRINT *print list* [*print delimiter*]

The PRINT statement causes data to be output at the terminal.

The data to be output is specified in a ***print list***. A ***print list*** consists of items (*numeric expressions, string expressions,* and ***print functions***) separated by commas or semicolons (which are ***print delimiters***). The *print list* may be followed by an optional ***print delimiter*** (comma or semicolon) which controls output formatting. If the ***print list*** is omitted, PRINT causes a skip to the next line.

A ***print function*** is one of the following:

- TAB (*numeric expression*) — tabs to column position
- LIN (*numeric expression*) — generates line feeds
- SPA (*numeric expression*) — spaces
- CTL (*numeric expression*) — outputs special device control commands

The contents of the ***print list*** are printed. If there is more than one item in the ***print list***, commas or semicolons must separate the items. The choice of a comma or semicolon affects the output format.

The output line is divided into five consecutive fields: four fields of 15 characters each and one field of 12 characters, for a total of 72 characters. When a comma separates items, each item is printed starting at the beginning of a field. When a semicolon separates items, each item is printed immediately following the preceding item. In either case, if there is not enough room left in the line to print the entire item, printing of the item begins on the next line.

The separator between items can be omitted if one or both of the items is a *literal string*. In this case, a semicolon is inserted automatically. For example PRINT "ABC" "DEF" would be printed in the same format as PRINT "ABC"; "DEF".

An X-OFF, carriage return, and line feed are output after each PRINT has executed, unless the output list is terminated by a comma or semicolon. In this case, the next PRINT statement begins on the same line.

If a *numeric expression* appears in the ***print list***, it is evaluated and the result is printed. Any variable must have been assigned a value before it is printed. A *literal string* is output "as is".

Numeric values are left justified in a field whose width is determined by the magnitude of the number. The smallest field is six characters. Numeric output format is discussed in detail below.

PRINT Statement (Cont)

Examples:

When items are separated by commas, they are printed in up to five fields per line; separated by semicolons, they directly follow one another. In the following example, the items are numeric, so each item is assigned a minimum of six characters.

```
10  LET A=B=C=D=E=15
20  LET A1=B1=C1=D1=E1=20
30  PRINT A, B, C1, C
40  PRINT A; B; C1; C; D; E; E; A1; D1; E1
50  PRINT A, B; C, D
60  END
RUN


 15              15              20              15
 15    15    20    15    15    15    15    20    20    20
 15              15    15        15


DONE
```

In the next example, a DIM statement is used to specify the number of characters in each string.

```
10  DIM B$[3], C$[3]
20  C$="ABC"
25  B$="ABC"
30  PRINT B$, C$
40  END
RUN


ABC             ABC

DONE
```

In the example below, the first PRINT statement evaluates and then prints three *numeric expressions*. The second PRINT skips a line. The third and fourth PRINT statements combine a *literal string* with a *numeric expression*. No fields are used in the print line for *literal string* unless a comma appears as a separator. The fourth PRINT statement prints output on the same line as the third because the third statement is terminated by a comma.

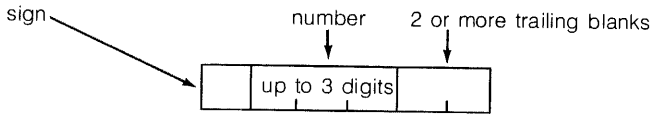
```
10  LET A=B=C=D=E=15
20  LET A1=B1=C1=D1=E1=20
30  PRINT A*B, B/C/D1+30, A+B
40  PRINT
50  PRINT "A*B =";A*B
60  PRINT "THE SUM OF A AND B IS";A+B
70  END
RUN


 225            30.05          30

A*B =  225
THE SUM OF A AND B IS 30

DONE
```

PRINT Statement (Cont)

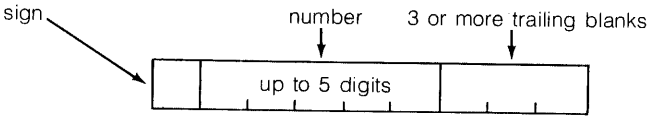
NUMERIC OUTPUT FORMATS

Numeric quantities are left justified in a field whose width is determined by the magnitude of the item. The width includes a position at the left of the number for a possible sign and at least one position to the right containing blanks. The width is always a multiple of three; the minimum width is six characters.

INTEGERS. An integer with a magnitude less than 1000 requires a field width of six characters:



An integer with a magnitude between 1000 and 32767 inclusive requires a field width of nine characters:



Examples of integers:

The integers below are less than 1000 and greater than −1000:

```
10  PRINT 1;999;30;-300;+295
20  END
RUN

 1      999    30    -300    295

DONE
```

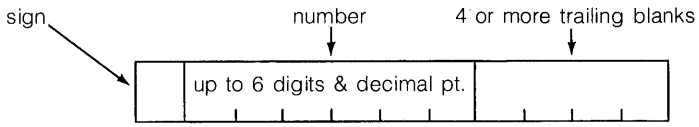
These integers are between 1000 and 32767 or between −1000 and −32767.

```
10  PRINT 1000;+32751;-32767;25678
20  END
RUN

 1000       32751    -32767      25678

DONE
```

PRINT Statement (Cont)

If an integer has a negative sign it is printed; a positive sign is not printed.

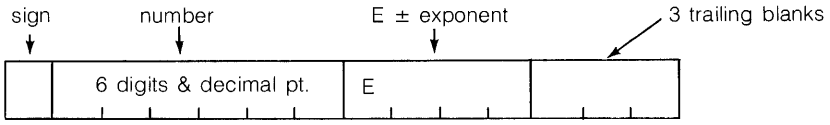
FIXED-POINT NUMBERS AND ALL OTHER INTEGERS. Fixed-point numbers and all other integers require a field width of 12 positions. If the magnitude of the number is greater than or equal to .09999995 and less than 999999.5, or is less than .1 but can be printed with six significant digits, the number is printed as a fixed-point number with a sign. Trailing zeros are not printed, but a decimal point is printed. The number is left-justified in the field with trailing blanks. The sign is printed only if it is negative.



Examples of fixed-point numbers:

```
10  PRINT 9.99999E+05;.1;.000044
20  END
RUN


 9.99999E+05     .1             .000044

DONE
```

ALL OTHER NUMBERS. Any number, integer or fixed-point, with a magnitude greater than the magnitude of the numbers presented above, is printed as a floating-point number using a total field width of 15 positions:



Examples of floating-point numbers:

```
10  PRINT 2.34568E+06;4.4E-06
20  END
RUN


 2.34568E+06     4.40000E-06

10  PRINT 2.34568E+07;4.4E-07
20  END
RUN

 2.34568E+07     4.40000E-07

10  PRINT 3.943E-05;2.57895E-05
20  END
RUN


 3.94300E-05     2.57895E-05
```

PRINT # Statement

General Form:

PRINT # *file number* [, *record number*]

or

PRINT # *file number* [, *record number*] ; END

or

PRINT # *file number* [, *record number*] ; *write list* [*print delimiter*]

The PRINT # statement is used to write data and control information to BASIC formatted and ASCII files.

A ***write list*** is any combination of *numeric expressions, string expressions* and *print functions* separated by ***print delimiters***. A ***print delimiter*** is a comma or a semicolon. In addition the last or only item in a ***write list*** can be the special print function: END.

BASIC FORMATTED FILE PRINTS

Serial (Sequential) Access Printing. Writing data to a file without specifying a *record number* causes the file to be filled serially, without respect to record boundaries. Successive prints cause data items to be stored one after the other beginning at the current position of the file pointer. A serial file print leaves the file pointer positioned immediately following the last item in the write list. Data written serially is usually read serially using successive file read (READ #) operations. Specifying END causes an End-Of-File (EOF) mark to be written on the file following the last item in the ***write list***. Writing an END leaves the file pointer positioned before the END.

Examples:

```
10 PRINT #1; A$,B1,N$(1,2)
20 PRINT #N+1; "TEXT",B$,X*Y,END
```

Direct (Random) Access Printing. Use of the *record number* after the *file number* in a PRINT # statement allows data to be stored in the record specified by the *record number*. Each direct file print causes items in the ***write list*** to be written at the beginning of the selected record. A direct file print leaves the file pointer positioned in the record immediately following the last item in the ***write list***.

In a direct file print operation, if the data in the ***write list*** exceeds the amount of space available in the record, an End-Of-File (EOF) condition is generated. In both direct and serial access, if printing beyond the physical end of the file is attempted, an End-Of-File (EOF) condition will occur. A trailing ***print delimiter*** (comma or semicolon) has no meaning for serial or direct access to BASIC formatted files. Print functions (TAB, LIN, SPA, and CTL) are ignored for serial or direct access to BASIC formatted files.

Examples:

```
10 PRINT #1,3;A$,B$,F2,E(3,4)
20 PRINT #N,R(3); A,B,"TEXT",END
```

PRINT # Statement (Cont)

Serial and direct file print statements can be used to write on the same file. A serial print following a direct print will write data immediately following the previous data.

The first record of a file is record number 1.

Numeric items occupy two words in a file. Strings occupy one word specifying the current *logical length* and one word for each two characters in the string. A string with an odd number of characters uses one word for the last character.

Serial File Print Example:

```
 10 FILES AFILE
 20 DIM A$(5)
 30 READ A,B,C,D,E,F,G,H,I,J
 40 DATA 100,200,300,400,500,600,700,800,900,1000
 50 LET A$="ABCDE"
 60 PRINT #1;"NUMBERS",A,B,C,D,E,F,G,H,I,J,END
 80 PRINT #1; A$
 90 PRINT #1;END
100 END
```

The string "NUMBERS" and the numbers 100 through 1000 are written onto the file number 1 (AFILE). An end-of-file mark is written following the last data item. Line 80 overlays the end-of-file mark previously written on AFILE with the string "ABCDE". Since the END is omitted, an end-of-record mark is automatically written after the string. Line 90 writes another end-of-file mark on the file.

Direct File Print Example:

```
 10 FILES AFILE
 20 READ A1,B1,C1,D1,E1
 30 DATA 1,2,3,4,5
 40 LET A$="A"
 50 FOR N=1 to 10
 60 LET B(N)=N+1
 70 NEXT N
 80 DIM B(10)
 90 PRINT #1,1;"START OF AFILE"
100 PRINT #1,2; 10,A1,B(1),B(2),B(3)
110 REM. .TWO RECORDS HAVE BEEN WRITTEN ON AFILE
120 PRINT #1,2; B1,C1,D1,E1
130 PRINT #1,1;A$
140 PRINT #1,3;END
150 REM. .THE THIRD RECORD OF AFILE IS AN END-OF-FILE
160 END
```

The first record of file number 1 contains a string value. The second record has five numeric items. The program writes four numeric items in the second record (overlaying the previous five items). A different string is written on record 1 and an end-of-file is written on record 3. Note that the records do not have to be written in the order they appear in the file.

PRINT # Statement (Cont)

ASCII FILE PRINTS

Printing to an ASCII file causes the data to be formatted in the same way as a print to your terminal. Numeric and string data items are written one per field (where a field is 15 print positions wide) if the items are separated with commas; they are written closely packed if separated with semicolons. The record length of an ASCII file determines the number of print fields per record. A print that generates a line longer than the record size of an ASCII file will result in truncation of the line by removing excess characters from the end of the line. Each print to an ASCII file results in a new line unless a trailing ***print delimiter*** exists in the ***write list***.

A record number cannot be specified when printing to ASCII files, but all of the ***print functions*** (TAB, LIN, SPA, and CTL) can be used and have the same effect as in a normal print to a terminal. Note that PRINT statements do not automatically print commas between ***write list*** items, and that ASCII file read operations require commas between data items (the same as if the data was input from your terminal). To meet this requirement, include a comma as a *literal string* (",") between items in the ***write list*** if the file is to be read later using a file read statement. Otherwise you can use the LINPUT # and CONVERT statements to read the records and extract numeric values from them.

Device status messages are sent to both your terminal and the system console to report device malfunctions or conditions requiring operator action. The messages DEVICE NOT READY, DEVICE ERROR, and ATTENTION NEEDED report device-specific conditions which cause suspension of the user operation. In the first two cases the system automatically restarts the operation after corrective action has been taken. For the ATTENTION NEEDED message the operator must use the AWAKE command to resume the user operation. The message READ/WRITE FAILURE is printed for non-recoverable errors. In this case the user operation is terminated in the same manner as a program error.

When using a line printer as an ASCII file, opening the file causes a page eject. Closing the file or using END in a print list has no effect on the line printer. The message LPn-DEVICE NOT READY means that someone must put the device on-line, clear a paper jam, insert new paper, or take other actions to ready the device. Any line whose printing is interrupted by a power failure will be automatically reprinted.

When using a paper tape punch as an ASCII file, the opening and closing of the file will punch leading and trailing feed frames respectively. Using END in the print list will cause 20 feed frames to be punched. Records are terminated with X-OFF, carriage-return, and line-feed unless the appropriate CTL function is employed to specify otherwise. The message PPn-DEVICE NOT READY is issued when the tape supply is low or the device is not turned on when opened. Any line whose punching was interrupted by a power failure will be terminated by a control-X, carriage-return, line-feed (to allow any later reading to ignore it) and then repunched.

PRINT # Statement (Cont)

When using the card reader as an ASCII file, opening it will cause pre-reading of the first two cards but closing it will have no effect (except to flush the images of any cards pre-read but not requested by the controlling program). A card beginning with a double colon (::) will be interpreted as an end-of-file condition. The message CRn-DEVICE NOT READY (device off-line, pick failure, hopper empty, etc.) requires the appropriate operator response. The message CRn-DEVICE ERROR signals a read check; the operator must replace the last-read card into the input hopper and ready the device. The message CRn-ATTENTION NEEDED indicates an illegal hole pattern detected in the last-read card; recovery requires some operator-user convention and use of the AWAKE command. A power failure will cause one of the above messages and need not be distinguished from any other conditions in terms of recovery action.

When using the paper tape reader as an ASCII file, opening it will cause a search for the first non-null character. Since this action is timed, it is important for the operator to avoid loading the tape with an excessive amount of leader (more than a few feed frames). Closing the file has no physical effect. A record consisting of 10 adjacent feed frames will be interpreted as an end-of-file condition. The message PRO-DEVICE NOT READY is issued when the device is opened before it is made ready (i.e., no tape in reader, too much leader, or device turned off). A power failure while the reader is in use will cause the message PRO-READ/WRITE FAILURE and its accompanying fatal error condition. Tape conventions are the same as for a terminal (i.e., carriage-return ends a record, line-feeds ignored, control-H deletes a character preceding, etc.).

When using a magnetic tape unit as an ASCII file, opening it causes a check to see if it is on-line and at load point. MTn-ATTENTION NEEDED will be issued if this is not true; the user program will be suspended until the operator corrects the problem and uses the AWAKE command (or the user elects to abort his program with the break key). When the file is closed an end-of-file mark is written if the last request issued was a write; in all cases the tape is rewound. It is also unloaded unless a rewind was already in progress when the file was closed. Including an END in a print list will write an end-of-file mark and then backspace over it, leaving the tape positioned at the end of the current file. The message MTn-ATTENTION NEEDED is issued to signal hardware malfunction or an attempt to read blank tape (in the latter case the tape is backspaced, leaving it positioned prior to the last record preceding the blank tape). An attempt to write on the file when no write ring is present, the occurrence of a power failure, the inability to read a record after three retries, or the inability to write a record after six retries will cause the message MTn-READ/WRITE FAILURE accompanied by a fatal error condition. Empty lines (null records) are simulated by writing a record containing two ASCII blanks. Records with an odd number of characters are padded with an ASCII blank when written (this will not be detected when such a record is read, the blank will be treated as part of the data). The end-of-file action will be taken (i.e., fatal error unless trapped with the IF END construct) when a read encounters an end-of-file mark, after a write with no intervening operation, or after the end-of-tape mark has been passed. The end-of-file action will also be taken if a write is issued after the end-of-tape mark has been passed. Note that the system will attempt to rewind and unload any tape belonging to an open file after recovery from a power failure; however, it will be unable to do so if power was also lost on the magnetic tape unit itself (in this case the system operator will have to manually rewind and unload the tape).

PRINT # Statement (Cont)

The control functions for the magnetic tape unit when used as an ASCII file are designed to facilitate use of both single file and multi-file reels of tape (but not multi-reel files). CTL(20) spaces forward until just prior to the first end-of-file mark encountered. If the tape was already so positioned, no action is taken. If the previous request was a write, an end-of-file mark is written and then backspaced over. Thus this control function always leaves the tape positioned at the end of the "current file", ready to extend it if so desired. Caution: if no end-of-file mark exists on the tape beyond the current position, no record exists beyond the end-of-tape mark, and the last request was not a write, then the unit will space over all of the tape and off the takeup reel. This condition cannot be diagnosed by the system and thus no diagnostic will appear (this problem can also occur with the next function described). CTL(21) spaces forward until just past the first end-of-file mark encountered (i.e., just prior to the first record, if any, of the next file). If the previous request was a write, an end-of-file mark is written and no further action is taken (i.e., the current file is completed and a new, empty file begun). If the end-of-tape mark has been passed when this request is made, the end-of-file action will be taken without any tape motion. CTL(22) backspaces until it just follows the first end-of-tape mark encountered (i.e., just prior to the first record of the current file), or to just following the load point indicator. If the previous request was a write, an end-of-file mark is written before backspacing. If the tape is already at the first record of a file, no action is taken. Thus this control function always leaves the tape positioned at the beginning of the current file. CTL(23) backspaces the tape until it just follows the second end-of-tape mark (or load point indicator) encountered (i.e., just prior to the first record of the preceding file). If the tape is currently positioned in the first file on the tape, the tape will be backspaced to the load point indicator and the end-of-file action is taken. If the previous request was a write, an end-of-file mark is written before backspacing. CTL(24) rewinds the tape to the load point indicator. If the previous request was a write, an end-of-file mark is written before rewinding.

Note that the system always ensures that an end-of-file mark appears after the last record written (unless that record crossed the end-of-tape mark, in which case the end-of-tape mark is treated as the end-of-file mark instead).

PRINT USING Statement

General Form:

PRINT USING *format part* [; *using list*]

The PRINT USING statement gives you more control over the format of your output data than the PRINT statement, but requires additional programming effort. The ***format part*** is either a ***format string*** (represented as a *literal string* or *string variable*) or a *statement number* referencing an IMAGE statement.

The optional ***using list*** is any combination of *numeric expressions, string variables*, or ***print functions*** (TAB, LIN, SPA, or CTL) separated by commas. Semicolons and trailing punctuation are not allowed. Commas in the ***using list*** serve as delimiters only; they have no formatting function. Note that *string variables,* not *string expressions*, are allowed.

A ***format string*** consists of an optional ***carriage control*** character and comma followed by one or more ***format specifications*** or ***groups of format specifications*** separated by a comma and/or slashes). Each ***format specification*** is either a *literal string* or an orderly combination of ***format characters***. ***Format strings*** may be up to 255 characters long. Blanks are ignored and lowercase characters are upshifted, except within literal strings.

Example:

```
PRINT USING 100; A$, (2+X), B$
```

CARRIAGE CONTROL CHARACTERS. One of the following optional ***carriage control*** characters may appear as the first non-blank character of a ***format string***:

+ suppress linefeed
− suppress carriage return
suppress carriage return and linefeed

If supplied, the ***carriage control*** character must be followed by a comma and at least one slash, ***format specification***, or ***group***. The specified action occurs at the completion of the PRINT USING statement. If the ***carriage control*** character is not supplied, the default action is an X-OFF, carriage return followed by a linefeed.

FORMAT CHARACTERS. ***Format specifications*** are comprised of orderly combinations of the following ***format characters***:

A reserves one character position in a string specification
D reserves one decimal digit position in a numeric specification
S defines the position of the sign character in a numeric specification
. defines the position of the decimal point in a numeric specification
E specifies floating point format in a numeric specification
X reserves one blank character position in a numeric, string, or literal specification.

An optional ***repetition factor*** (replicator), between 1 and 255 inclusive, may precede an A, X, or D (e.g. 3A is equivalent to AAA).

PRINT USING Statement (Cont)

GROUPS. A ***group*** of one or more ***format specifications*** may be enclosed in parentheses which must be preceded by a ***repetition factor*** between 1 and 255 inclusive (e.g. 2(/AX,D/) is equivalent to /AX,D///AX,D/). Within the parentheses, the specifications must be separated by commas or slashes and the ***group*** *must be set off from other specifications by a comma or slashes, just as if it were a single specification.* ***Groups*** *can be nested two levels deep.*

REPETITION FACTORS (REPLICATORS). An unsigned positive integer between 1 and 255 inclusive must precede the left parenthesis (of a ***group*** and may optionally precede any X, D, or A in a ***format specification***. It indicates the number of times the following character or group is to be repeated. Leading zeros are allowed.

EXECUTION OF THE PRINT USING STATEMENT. Execution of the PRINT USING statement commences by examining the *format string*. The ***carriage control*** character, if present, is noted for termination processing, then each ***format specification*** is examined.

If the specification is either a string or a numeric specification, the next item from the ***using list*** is printed according to the specification. If the ***using list*** has already been exhausted or is not present, the statement terminates. If the item does not agree with the specification (i.e. string vs. numeric), an error message is printed and the program execution terminates.

If the specification is literal, the specified number of blanks (or the contents of the literal string) is simply printed; the ***using list*** is not examined.

If the end of the ***format string*** is reached before the end of the ***using list***, processing continues from the beginning of the ***format string*** but after the optional ***carriage control*** character (if the ***format string*** contains no string or numeric specifications, the statement terminates).

When all items from the ***using list*** have been printed the statement terminates (any remaining literal specifications are processed if the end of the ***format string*** has not been reached for the first time). Termination consists of printing an X-OFF, carriage return, and linefeed, modified by the ***carriage control*** character.

If the ***format string*** is empty or contains only blanks, output consists of only an X-OFF, carriage return, and linefeed.

FORMAT SPECIFICATIONS. There are three catagories of ***format specifications***: those containing at least one D (numeric), those containing at least one A (string), and those consisting of either a literal string or all X's (literal); A's and D's may not appear in the same ***format specification***. Numeric specifications are further classified as integer (no decimal point or exponent), fixed-point (contain a decimal point but no exponent), and floating-point (contains an exponent and possibly a decimal point). Table 10-4 describes the output formats that result from each type of ***format specification***.

DELIMITERS. ***Format specifications*** must be separated with either a comma or a slash (/). A comma serves only as a delimiter whereas a slash also causes an X-OFF, carriage return, and linefeed to be output. Commas may appear only between ***format specifications*** and may not be adjacent. Slashes may appear before, after, and in place of format specifications as well as between them, therefore slashes may be adjacent to commas. Multiple slashes (///) are allowed but a slash may not be preceded by a ***repetition factor***.

PRINT USING Statement (Cont)

Format Specification Examples

TYPE	COMBINATION RULES	EXAMPLES
INTEGER	Any combination of X's and D's is allowed, but at least one D must be present and only one S is allowed. This specification must match a number in the print list. The number is rounded to an integer and printed right-justified. Although the requested number of digits will be printed, only six are significant. If there is not enough room in the field for the number (i.e. the number of digits is greater than the number of D's in the format), then the value is printed on a separate line in a floating point format (SD.5DE). If an S precedes all D's, the sign is printed immediately preceding the first digit of the number. If an S appears after the first D, the sign is printed at the location of the S. If an S is not included in the format, then an extra D should be provided if the value is negative. When the value is negative, the - sign always precedes the most significant digit and a space must be provided with an extra D to prevent overflow. Blanks may be combined with carriage control to cause overprinting. For example, large numbers can be printed with blanks in the positions reserved for commas (e.g. $10,937).	FORMAT SPEC / VALUE / OUTPUT DDDD, 4D, 2DDD, 2D2D, 2(2D) } equivalent — 1234 — 1234 4D — 1234.8 — 1235 S4D — 1234 — +1234 4DS — 1234 — 1234+ 5D — 1234 — ^1234 5D — -1234 — -1234 DX3D — 1234 — 1^234 DS3D — 1234 — 1+234 S10D — 1234 — ^^^^^^+1234
FIXED-POINT	Any combination of D's and X's to the left and right of the decimal point is allowed, but at least one D must be present and only one S and one "." are allowed. For this specification, the next item in the print list must be a numeric quantity. The digits to the right of the decimal point are rounded to fit the field. Leading zeros to the left are suppressed, but trailing zeros are printed. If the number to be printed has no digits to the left of the decimal point but D's are provided to the left, then a zero will be printed in the rightmost D on the left side. If an S is provided to the left, it is moved to the right through D's and X's until it comes to the first non-blank character. If an S is not provided and the number is negative, 1. no D's to the left causes overflow; 2. one D to the left will be used for the minus sign and the leading zero will be dropped; or 3. two or more D's to the left will have the minus sign and zero printed in the two rightmost D positions.	DDD.DDDD, DDD.4D, 3D.4D, 3D.DDDD } equivalent — 465.465 — ^465.4650 4D.2D — 465.465 — ^465.47 4D.3D — -465.465 — -465.465 SDD2D.D — 465.465 — ^+465.5 S2D.4D — .465 — ^+0.4650 S.4D — .465 — +.4650 D.4D — -.465 — -.4650 2D.4D — -.465 — -0.4650

Format Specification Examples (Continued)

TYPE	COMBINATION RULES	EXAMPLES		
		FORMAT SPEC	VALUE	OUTPUT
FLOATING-POINT	Any legal INTEGER or FIXED-POINT format specification may be followed by an E. The E signifies a four character field consisting of an "E" followed by "+" or "–" and two decimal digits. This format is useful for numbers that are very large or very small. For example, .00005 = .5 x 10^{-4} = .5E-4. When X's follow the E they cause blanks to be printed between the E and the exponent sign. The output value from the print list is multiplied or divided by 10, the number of times necessary to fit the value into the field. It is then rounded from the right, and the exponent is adjusted to match the number of multiplications or divisions. If the format allows for more digits than there are significant digits in the output value, two rules are followed: 1. If there are more than 6 D's on the right side of the decimal point, the leftmost digit is printed in the first D (if any) to the left of the decimal point or the first D to the right of the decimal point; extra D's beyond 7 on the right are filled with non-significant digits. In the following examples, the arrow indicates the position of the leftmost digit printed: DD.40D XX.DD40D 40DD.40D ↑ ↑ ↑ 2. If there are less than 7 D's on the right side of the decimal point, the leftmost digit is printed in the seventh D position from the right (or the leftmost if there are not 7). In the following examples, the arrow indicates the position of the leftmost digit printed: 6DDDD.DDDD DD.DD D.6D ↑ ↑ ↑	SDXE DDDD.DDE S5DX.X5DEX SD.5DE S.10DE3X	$4.82716X10^{21}$	+5^E+21 4827.16E+18 ^^^+48^.^27159E^+20 +4.82716E+21 +.4827159382E^^^+22
STRINGS	Any combination of A's and X's. Strings are left-justified in the field. Leftover spaces are filled with blanks. If the string contains more characters than the specification allows, characters are truncated from the right.	AAAA 2A2A 2(2A) 4A	equivalent	ABCDEF	ABCD
		6A 8A 2X6A AXAXAXAXAXA			ABCDEF ABCDEF^^ ^^ABCDEF A^B^C^D^E^F
LITERAL	A specification consisting only of X's or any *literal string.*	XX4X		none	^^^^^^
	If the *format string* is represented as a *literal string* and contains *literal string* specifications, the outer quote marks must be represented by using the *extend string literal* convention (i.e. '34).	Literal^string1		none	Value of Literal String

PRINT USING Statement (Cont)

Print Using Format Errors

FATAL ERRORS

These errors cause termination of the program. An error message is printed along with the offending specification.

1. Replicator is outside the range $1 \leq n \leq 255$.
2. D, S, E, or . occurs in a string specification.
3. A occurs in a numeric specification.
4. Characters other than A, X, D, S, E, /, or . occur in any specification other than a literal.
5. Adjacent commas.
6. More than two levels of parentheses.
7. No D in a numeric specification.
8. An S in a blank specification.
9. String expression encountered for a numeric format specification.
10. Two or more E's, S's, or .'s in a specification.
11. Replicator not followed by (, D, X, or A.
12. Literal string not set off by delimiters and enclosed in quotes.
13. Parentheses used without a replicator.
14. Referenced statement is not IMAGE.
15. Numeric data encountered by a string format.
16. Leading or trailing commas in group or format string (i.e. null group or missing format specification).

NON-FATAL ERRORS

These errors do not terminate the program.

1. String specification too small for the string data causes the string to be truncated from the right.
2. Field too small for number. Causes the number to be printed on the next line using SD.5dE format. Printing resumes on the next line.

PURGE Statement

General Form:

PURGE ***return variable , file designator***

The PURGE statement deletes BASIC formatted or ASCII files from the system.

The *return variable* returns the status of the purge operation. The values returned and their meaning are as follows:

Return Value	Meaning
0	File successfully purged
1	File is busy and cannot be purged
2	File is not accessible
3	No such file

The *file designator* is a *source string* whose value is a *file name*.

The PURGE statement can be used to dissociate nonsharable devices from the name used in a FILE command. If the ASCII file is a disc file, the space it occupied is returned to the system.

A locked or private program saved in a group account library having the FCP (File Create/ Purge) capability can be executed or chained-to in order to purge locked BASIC formatted files in group member accounts having the PFA (Program/File Access) capability. These files may be specified by appending ".idcode".

Examples:

```
10 PURGE N, "MYFILE"
20 PURGE N, "HERFIL.G707"
30 PURGE N, A$
```

READ Statement

General Form:

READ *read variable list*

The READ statement reads string and numeric values from DATA statements.

A ***read variable list*** is one or more ***read variables*** separated by commas. A ***read variable*** can be a *destination string* or a *numeric variable*.

Reading begins with the first item in the first DATA statement in the program. Subsequent reads continue with the last item not read. The type of ***read variable*** must agree with the data type in the DATA statement. You can use the TYP function to determine the next data type in a DATA statement.

Examples:

```
10 READ A,C(33), F1$(20,40)
20 READ G(I,J/2), Z$
```

READ # Statement

General Form:

READ # ***file number***[, record number][; read variable list]

The READ # statement reads data from BASIC formatted or ASCII files.

A serial file read statement (no *record number* specified) reads items from a file specified by *file number* into variables specified in the ***read variable list***. A ***read variable list*** is one or more ***read variables*** separated by commas. A ***read variable*** is a *numeric variable* or a *destination string*. The first item read is the item following the current position of the file pointer. Record boundaries are ignored and the items read can begin in the middle of one record and end in the middle of another. If the *read variable list* is omitted, no action is taken.

A direct file read statement (*record number* specified) reads items from a specific record in a file into variables of the ***read variable list***. If a record number is specified that is outside the range for the named file, an end-of-file condition occurs. If the ***read variable list*** is omitted, a direct file read moves the pointer to the beginning of the specified record, but does not read data.

The destination for string data must be a *destination string* and the destination for numeric data must be a *numeric variable*. If the data does not match the type of the destination variable, an error will occur. It is possible to check the type of the next data item with the TYP function described elsewhere in this section.

In both direct and serial file access, an attempt to read beyond a logical or physical end-of-file will cause the end-of-file condition to occur. Unless an IF END statement is used to transfer control to another statement in the program, an error will occur terminating the program.

READ # Statement (Cont)

Serial File Read Examples:

```
10  FILES AFILE,BFILE
20  DIM A$[5],X$[10],Y$[10],C$[15]
30  DIM B[10]
40  MAT  READ B
50  DATA 100,200,300,400,500,600,700,700,900,1000
60  LET A$="ABCDE"
70  PRINT #1;"ARRAY B"
74  FOR J=1 TO 10
75  PRINT #1;B[J]
76  NEXT J
80  PRINT #2;A$,B[3],B[6]
90  PRINT #2; END
100  PRINT #1;"END OF ARRAY"
110  READ #1,1
120  READ #2,1
130  READ #1;X$,A1,B1,C1,D1,E1
140  PRINT X$,LIN(1),A1,B1,C1,D1,E1
150  READ #1;A2,B2,C2,D2,E2
160  PRINT A2,B2,C2,D2,E2
170  READ #2;Y$,A,B
180  PRINT Y$,A,B
190  READ #1;C$
195  PRINT C$
200  READ #1;X
210  REM..ATTEMPT TO READ END-OF-FILE CAUSES TERMINATION
220  END
RUN

ARRAY B
 100            200            300            400            500
 600            700            700            900            1000
ABCDE           300            600
END OF ARRAY

END-OF-FILE/END OF RECORD  IN LINE  200
```

READ # Statement (Cont)

After data is written on files 1 and 2 with the print statements in lines 70 through 100, and the pointer is restored to the start of each file in lines 110 and 120, the data is ready to be read. The first six items in file 1 are read in line 130. The next five items are read in line 150. The three items written on file 2 are read in line 170. The PRINT statements are inserted to test the accuracy of the reads and the previous writes. A string item remains in file 1 and is read in line 190. Line 200 attempts to read an end-of-file mark causing the message: END OF FILE/END OF RECORD IN LINE 200 to be printed.

Direct File Read Examples:

```
5  FILES AFILE
10  DIM C[5],D[5]
20  DIM A$[20],X$[20]
25  MAT C=CON
30  READ A,B,C
40  DATA 1,2,3
50  PRINT #1,2;A,B,C,"NUMBERS"
60  PRINT #1,1;"ARRAY C"
70  FOR I=1 TO 5
80  PRINT #1;C[I]
90  NEXT I
100  READ #1,1;X$
110  FOR I=1 TO 5
120  READ #1;D[I]
130  NEXT I
140  PRINT X$
150  MAT  PRINT D
160  READ #1,2;D,E,F,A$
170  PRINT A$,D;E;F
180  END
RUN


ARRAY C
 1

 1

 1

 1

 1

NUMBERS           1       2       3

DONE
```

ASCII File Read Operations

Reading from an ASCII file follows the same rules as the INPUT statement. The data is read as if it were being input at your terminal. Refer to PRINT # for a discussion of ASCII file access.

REC Function

General Form:

REC (*numeric expression*)

The REC function returns the current record number being accessed in the file specified.

The *numeric expression* is evaluated and rounded to an integer. This integer is used as the *file number*. The REC function cannot be used with ASCII files. If attempted the program will be terminated with an error. Specification of an invalid *file number* will also result in an error.

Examples:

```
20 PRINT "I AM NOW AT RECORD " REC(N) "IN FILE " N
30 R=REC(N+1)
```

REM Statement

General Form:

REM [***remark***]

The REM statement allows you to add comments to your program.

The optional ***remark*** may be any series of characters. The REM statements are not executed.

Examples:

```
200 REM****ITEM SEARCH SUBROUTINE****
300 REM This program is used to sort data by value
400 REM THIS PORTION OF THE PROGRAM IS ONLY
401 REM EXECUTED IF THE DATA IS INVALID
402 REM AND THE ERROR DETECT FLAG IS SET
500 REMARK — "ARK" are the first three characters of this REM statement
```

RESTORE Statement

General Form:

RESTORE [***statement number***]

The RESTORE statement resets the program's data pointer to the first item in a DATA statement.

The optional *statement number* indicates a specific DATA statement. If no *statement number* is given, the pointer is reset to the first data item in the first DATA statement in the program.

If the *statement number* does not reference a DATA statement, the next READ or MAT READ statement will cause the pointer to be moved to the first DATA statement following the referenced statement. If there is no following DATA statement, the attempted read operation will terminate the program with an error.

Examples:

```
20 RESTORE 30
30 RESTORE
```

RND Function

General Form:

RND (*numeric expression*)

The RND function is a numeric valued function which returns a pseudo random number in the range $0 \leq R < 1$.

The number is not truly random but is calculated from an initial or "seed" number. Each succeeding number in the sequence is then calculated from the previous number. An initial seed number (based on the date and time of day) is assigned to each port on the system every time the system is started. The sequence may be altered by specifying a new seed. This allows you to repeat given sequences of random numbers.

The *numeric expression* is evaluated and used to obtain the next random number based on the last number or to change the last number and begin a new sequence as follows:

Value of the *Numeric Expression*	**Result**
less than 0	A pseudo random number is returned that is calculated from the *numeric expression*.
greater than or equal to 0	A pseudo random number is returned that is calculated from the last number generated.

In order to generate a repeatable sequence of pseudo random numbers, you should first use the RND function with a *numeric expression* that has a value <0, and then continue using the function with a *numeric expression* that has a value > = 0. To repeat the sequence, use the value from the initial call.

Examples:

```
100 N3=RND(-5.55)
110 D  =RND(F)**  10
```

SGN Function

General Form:

SGN (*numeric expression*)

The SGN function is a numeric valued function which returns a number indicating the sign of the *numeric expression*. If the sign of the *numeric expression* (after evaluation) is positive, the value returned is 1. If the sign is negative, the value returned is −1. If the numeric expression evaluates to zero, the value returned is 0. For instance, SGN (−3.14) = −1, SGN (943.2) = 1, and SGN (0) = 0.

Examples:

```
1200 A4 = SGN (P2)
1210 A(I) = SGN (A2+A(I−1))
```

SIN Function

General Form:

SIN (*numeric expression*)

The SIN function is a numeric valued function which returns the sine of the *numeric expression*. The *numeric expression* is interpreted as being in radians. If the absolute value of the *numeric expression* exceeds approximately 102900, your program will be terminated with an error.

Example:

```
440 LET S(I) = SIN(3.1415/I)
```

SPA Function

General Form:

SPA (*numeric expression*)

The SPA function is used in a PRINT, PRINT#, PRINT USING, MAT PRINT, MAT PRINT#, or MAT PRINT USING statement to print the number of blank characters specified by the *numeric expression*.

The *numeric expression* is evaluated and rounded to an integer. The SPA function is ignored when the *numeric expression* is zero or negative. If the number of blanks will not fit on the current line, or if *numeric expression* evaluates to a value greater than 71, an X-OFF, carriage return, and line feed are generated. PRINT USING and MAT PRINT USING statements using the SPA function do not have a line length limit of 71 characters. (Line printers and other ASCII devices can have more than 72 characters.) SPA is ignored for file prints (PRINT#, MAT PRINT#) to BASIC formatted files, but is allowed for ASCII files. Note that a comma after a SPA function is treated as a semicolon (see ***print delimiter*** under PRINT).

Example:

```
10 PRINT A$, SPA(10), B$, SPA(X+LEN(C$), D$
```

SQR Function

General Form:

SQR (*numeric expression*)

The SQR function is a numeric valued function which returns the square root of the *numeric expression*. The *numeric expression* must evaluate to a number greater than or equal to 0; otherwise, your program will terminate with an error.

Examples:

```
932 LET S2 = SQR(2)
940 PRINT SQR (B**2 + C **2)
```

STOP Statement

General Form:

STOP

The STOP statement terminates execution of the program and returns control to the system. The STOP statement may occur at any point in the program except the last statement which must be END.

Example:

```
300 STOP
```

SYSTEM Statement

General Form:

SYSTEM *return variable , source string*

or

SYSTEM *destination string , source string*

The SYSTEM statement allows you to execute some operating system commands from a running program.

The command is given in the *source string* and must be in the same form that would be used to enter the command normally. The *OUT=file name* construct allowed in some commands is not allowed in the SYSTEM statement. Any lower case letters used in the *source string* are shifted to upper case.

Commands which can be used with the first form of the SYSTEM statement are BYE, ECHO, MESSAGE, FILE, PROTECT, LOCK, PRIVATE, UNRESTRICT, MWA, and SWA. The *return variable* is set to 0 if the command is executed and 1 if it is not.

A locked or private program saved in a group account library having the FCP (File Create/ Purge) capability can be executed or chained to in order to specify a locked BASIC formatted file MWA using the SYSTEM statement. The file must belong to a group member having the PFA (Program/File Access) capability. The group account library must have the MWA (Multiple Write Access) capability.

Example:

```
SYSTEM R, "MWA – GRTH.C402"
```

Commands which can be used with the second form of the SYSTEM statement are TIME, CATALOG, GROUP, LIBRARY, and LENGTH. These commands would normally produce output on your terminal. Only the first line of this output (less any heading) will be returned in the *destination string*. Note that for the CATALOG, GROUP, and LIBRARY commands you can enter start print parameter (refer to the descriptions for these commands in Section 10).

In the example given in line 30 below, the first line of actual data, starting with the first entry equal to or greater than "T" will be returned as the value of A$.

Examples:

```
10 SYSTEM R0, "FILE-LPR,LP0"
20 SYSTEM A$, B$
30 SYSTEM A$, "CAT-T"
40 SYSTEM P0$(1,4),"TIM"
```

TAB Function

General Form:

TAB(*numeric expression*)

The TAB function is used in print operations to move the current print position to a specified column.

The TAB function can be used in PRINT, PRINT#, PRINT USING, MAT PRINT, MAT PRINT #, and MAT PRINT USING statements. The *numeric expression* is evaluated and rounded to an integer to obtain the destination column. Print columns are numbered 0 to 71 (note that line printers and other ASCII devices can have more than 72 columns). If the *numeric expression* evaluates to a column position lower than the current print position, the TAB function is ignored. If the *numeric expression* evaluates to more than 71, the print position is moved to the beginning of the next line (RETURN) unless this occurs in a PRINT USING or MAT PRINT USING statement. In PRINT USING and MAT PRINT USING no X-OFF, carriage return, line feed will be generated if the TAB argument exceeds 71.

The TAB function is ignored for file prints (PRINT # and MAT PRINT #) to BASIC formatted files but is allowed for ASCII files.

Commas following TAB functions in print statements are treated as semicolons (refer to the discussion ***print delimiter*** under the PRINT statement).

Examples:

```
10 PRINT A,B$(1,10),TAB (40)
20 PRINT TAB(A+B),N,TAB(C),M
```

TAN Function

General Form:

TAN (*numeric expression*)

The TAN function is a numeric valued function that returns the tangent of the *numeric expression.*

The *numeric expression* is interpreted as being in radians. If the absolute value of the *numeric expression* exceeds approximately 51500, your program will be terminated with an error.

Example:

```
14 A3 = TAN(3.1415/P2)
```

TIM Function

General Form:

TIM (*numeric expression*)

The TIM function is a numeric valued function which returns the current second, minute, hour, day, or year.

The *numeric expression* is evaluated and rounded to an integer. This integer is used to specify which of the time values is to be returned as follows:

Value of Expression	Time Returned
0	current minute (0-59)
1	current hour (0-23)
2	current day (1-366)
3	current year (0-99)
4	current second (0-59)

Note the difference between the TIM function and the TIME command. The TIME command is used outside a program and gives the current console time used for that port, the total time used to date for that account, and the total time permitted for that account. The TIM function must be used within a program and returns the current time based on the system clock.

Examples:

```
20 IF TIM(0)-A = 15 THEN 9000
30 A3= TIM(B)
40 PRINT "SECOND";TIM(4);"MINUTE";TIM(0);"HOUR";TIM(1);"DAY";TIM(2)
50 PRINT TIM(2*X/A-2)
```

TRN Function

(Refer to MAT. . .TRN)

TYP Function

General Form:

TYP (*numeric expression*)

The TYP function is a numeric valued function that returns the data type of the next sequential item in a file or data statement.

The *numeric expression* is evaluated and rounded to an integer. The absolute value of the integer must be a valid *file number* or 0. If it is not, the program will be terminated with an error.

When the TYP function is used to test data in the file indicated by the *numeric expression* it will return a value indicating that the next item in the file is a number, string, End-Of-File mark, or End-Of-Record mark. If the numeric expression evaluates positive, End-Of-Record marks are ignored. This allows the TYP function to be used with serial files.

When used to test data contained in a program DATA statement, the *numeric expression* must have a value of 0.

Return values for various values of the *numeric expression* are as follows:

Return Value	Expression < 0 (file test)	Expression > 0 (file test)	Expression = 0 (DATA statement test)
1	number	number	number
2	string	string	string
3	end of file	end of file	end of data
4	end of record	—	—

Note that ASCII files will return a value of 2 only.

Examples:

```
10 A = TYP(-1)
20 GOTO TYP(-X) OF 40,50,6070
30 PRINT TYP(X*2)
```

UNLOCK Statement

General Form:

UNLOCK # *file number* [, *return variable*]

The UNLOCK statement clears a file status flag that has been previously set with a LOCK statement (refer to the LOCK statement).

The UNLOCK statement should be used to release a file as soon as the file access operation is complete to allow others using the locking technique to access the file. If the optional *return variable* is used you can check the values returned to determine the result of the unlocking operation. A list of return values and their meaning is given below.

Return Value	Meaning
0	File successfully unlocked
1	File already unlocked (either by this or another program)
2	File number invalid

The UNLOCK statement does not affect LOCK statements in other programs. Execution of an UNLOCK statement leaves the file pointer positioned at the end of the last accessed record in the file.

Examples:

```
10 UNLOCK #3,Z0
20 UNLOCK # N
```

UPDATE Statement

General Form:

UPDATE # *file number* ; *numeric expression*

or

UPDATE # *file number* ; *source string*

The UPDATE statement replaces the next sequential data item in the file referenced by the *file number* with the value of either the *numeric expression* or the *source string*.

The data type of the new item must match the type (numeric or string) of the item being replaced. The *file number* must refer to a BASIC formatted file. The UPDATE statement cannot be used with ASCII files. The *source string* must be the same size as the file string being replaced. If the *source string* is shorter than the old string, it will be padded on the right with blanks. If the *source string* is longer than the old string it will be truncated from the right to fit.

End-Of-Record (EOR) marks are ignored. The next sequential item after the EOR mark will be updated. Attempting to update an End-Of-File (EOF) mark will result in an end of file condition. You must have write access to the file to use the UPDATE statement.

Examples:

```
10 UPDATE #3 ; A$
20 UPDATE #1 ; "JIM"
30 UPDATE #7 ; Z1$(3,20)
40 UPDATE #2 ; A(N,M)
50 UPDATE #4 ; X*12/52
```

UPS$ Function

General Form:

UPS$ (*source string*)

The UPS$ function returns a string equivalent of the *source string* with all characters shifted to upper case.

Only the 26 lower case alphabetic characters (a to z) are effected. For example, UPS$("abc%12") would return the string "ABC%12".

Examples:

```
200 PRINT UPS$(G$)
300 Q$=UPS$(R$(j,k))
```

ZER Function

(Refer to MAT. . .ZER)

USING THE ASCII CHARACTER SET

APPENDIX A

Most terminals used with the 2000 Access System represent characters in ASCII (American Standard Code for Information Interchange) code. This code is used to store all string data. The ASCII character set is given in table A-1. An alternate code used by the IBM 2741 Communication terminal is discussed in Appendix D.

The 128 character set in table A-1 is ranked according to the decimal equivalent of the code. This decimal value is used in determining the relative value of strings. Comparison of strings is done character by character. Note that space (blank) characters actually have a value (32). This fact must be taken into consideration when comparing strings of different length.

The decimal equivalent of a character can be used as an extended string literal if preceded by an apostrophe ('). A discussion of literal strings is contained in section XI.

Note that in table A-1, characters with values 0 through 32 do not have a graphic or printing character. They are typically used to perform a variety of terminal and communication operations. Characters with values between 0 and 63 can be generated using an alternate character combination. A bell code for example can be generated by holding the control key and pressing the G key (G^c). This technique can be used to produce ASCII characters that may not appear as separate keys on your terminal. This is important since the character set available on terminal keyboards may vary.

When using the RJE facility for transmission of data to IBM or CDC host systems you should refer to table A-2 for a list of valid data characters. Note that some characters available on the 2000 Access System are not available on the IBM and CDC systems. In addition some character codes generate different graphics on different systems.

When reading cards using a BASIC program, refer to table A-2 for a list of valid character codes. Note that the lower case alphabetic characters cannot be read by a BASIC program.

Table A-1. ASCII Character Set

DECIMAL VALUE	GRAPHIC	COMMENTS	ALTERNATE CHARACTER	DECIMAL VALUE	GRAPHIC	COMMENTS
0		Null	@[c]	64	@	Commercial at
1		Start of heading	A[c]	65	A	Uppercase A
2		Start of text	B[c]	66	B	Uppercase B
3		End of text	C[c]	67	C	Uppercase C
4		End of transmission	D[c]	68	D	Uppercase D
5		Enquiry	E[c]	69	E	Uppercase E
6		Acknowledge	F[c]	70	F	Uppercase F
7		Bell	G[c]	71	G	Uppercase G
8		Backspace	H[c]	72	H	Uppercase H
9		Horizontal tabulation	I[c]	73	I	Uppercase I
10		Line feed	J[c]	74	J	Uppercase J
11		Vertical tabulation	K[c]	75	K	Uppercase K
12		Form feed	L[c]	76	L	Uppercase L
13		Carriage return	M[c]	77	M	Uppercase M
14		Shift out	N[c]	78	N	Uppercase N
15		Shift in	O[c]	79	O	Uppercase O
16		Data link escape	P[c]	80	P	Uppercase P
17		Device control 1 (X-ON)	Q[c]	81	Q	Uppercase Q
18		Device control 2	R[c]	82	R	Uppercase R
19		Device control 3 (X-OFF)	S[c]	83	S	Uppercase S
20		Device control 4	T[c]	84	T	Uppercase T
21		Negative acknowledge	U[c]	85	U	Uppercase U
22		Synchronous idle	V[c]	86	V	Uppercase V
23		End of transmission block	W[c]	87	W	Uppercase W
24		Cancel	X[c]	88	X	Uppercase X
25		End of medium	Y[c]	89	Y	Uppercase Y
26		Substitute	Z[c]	90	Z	Uppercase Z
27		Escape	[[c]	[1] 91	[	Opening bracket
28		File separator	\[c]	[2] 92	\	Reverse slant
29		Group separator	][c]	[1] 93	]	Closing bracket
30		Record separator	^[c]	[1] 94	^	Circumflex
31		Unit separator	_[c]	[2] 95	_	Underscore
32		Space (Blank)	`[c]	96	`	Grave accent
[1]33	!	Exclamation point	a[c]	97	a	Lowercase a
34	"	Quotation mark	b[c]	98	b	Lowercase b
35	#	Number sign	c[c]	99	c	Lowercase c
36	$	Dollar sign	d[c]	100	d	Lowercase d
37	%	Percent sign	e[c]	101	e	Lowercase e
38	&	Ampersand	f[c]	102	f	Lowercase f
39	'	Apostrophe	g[c]	103	g	Lowercase g
40	(	Opening parenthesis	h[c]	104	h	Lowercase h
41	)	Closing parenthesis	i[c]	105	i	Lowercase i
42	*	Asterisk	j[c]	106	j	Lowercase j
43	+	Plus	k[c]	107	k	Lowercase k
44	,	Comma	l[c]	108	l	Lowercase l
45	-	Hyphen (Minus)	m[c]	109	m	Lowercase m
46	.	Period (Decimal)	n[c]	110	n	Lowercase n
47	/	Slant	o[c]	111	o	Lowercase o
48	0	Zero	p[c]	112	p	Lowercase p
49	1	One	q[c]	113	q	Lowercase q
50	2	Two	r[c]	114	r	Lowercase r
51	3	Three	s[c]	115	s	Lowercase s
52	4	Four	t[c]	116	t	Lowercase t
53	5	Five	u[c]	117	u	Lowercase u
54	6	Six	v[c]	118	v	Lowercase v
55	7	Seven	w[c]	119	w	Lowercase w
56	8	Eight	x[c]	120	x	Lowercase x
57	9	Nine	y[c]	121	y	Lowercase y
58	:	Colon	z[c]	122	z	Lowercase z
59	;	Semicolon	{[c]	[2]123	{	Opening (left) brace
60	<	Less than	|[c]	[2]124	|	Vertical line
61	=	Equals	}[c]	[2]125	}	Closing (right) brace
62	>	Greater than	~[c]	[2]126	~	Tilde
63	?	Question mark	delete[c]	127		Delete

Notes: 1. The equivalent EBCDIC character uses a different graphic.
2. No equivalent character exists in EBCDIC.

Table A-2. EBCDIC Character Set for Use with RJE

ASCII	GRAPHIC IBM	CDC	PUNCH CODE (029)	COMMENTS
A	Same as ASCII	Same as ASCII	12-1	
B	Same as ASCII	Same as ASCII	12-2	
C	Same as ASCII	Same as ASCII	12-3	
D	Same as ASCII	Same as ASCII	12-4	
E	Same as ASCII	Same as ASCII	12-5	
F	Same as ASCII	Same as ASCII	12-6	
G	Same as ASCII	Same as ASCII	12-7	
H	Same as ASCII	Same as ASCII	12-8	
I	Same as ASCII	Same as ASCII	12-9	
J	Same as ASCII	Same as ASCII	11-1	
K	Same as ASCII	Same as ASCII	11-2	
L	Same as ASCII	Same as ASCII	11-3	
M	Same as ASCII	Same as ASCII	11-4	
N	Same as ASCII	Same as ASCII	11-5	
O	Same as ASCII	Same as ASCII	11-6	
P	Same as ASCII	Same as ASCII	11-7	
Q	Same as ASCII	Same as ASCII	11-8	
R	Same as ASCII	Same as ASCII	11-9	
S	Same as ASCII	Same as ASCII	0-2	
T	Same as ASCII	Same as ASCII	0-3	
U	Same as ASCII	Same as ASCII	0-4	
V	Same as ASCII	Same as ASCII	0-5	
W	Same as ASCII	Same as ASCII	0-6	
X	Same as ASCII	Same as ASCII	0-7	
Y	Same as ASCII	Same as ASCII	0-8	
Z	Same as ASCII	Same as ASCII	0-9	
0	Same as ASCII	Same as ASCII	0	
1	Same as ASCII	Same as ASCII	1	
2	Same as ASCII	Same as ASCII	2	
3	Same as ASCII	Same as ASCII	3	
4	Same as ASCII	Same as ASCII	4	
5	Same as ASCII	Same as ASCII	5	
6	Same as ASCII	Same as ASCII	6	
7	Same as ASCII	Same as ASCII	7	
8	Same as ASCII	Same as ASCII	8	
9	Same as ASCII	Same as ASCII	9	
blank	Same as ASCII	Same as ASCII	no punch	space
+	Same as ASCII	Same as ASCII	12-8-6	plus
-	Same as ASCII	Same as ASCII	11	minus (hyphen)
*	Same as ASCII	Same as ASCII	11-8-4	asterisk
/	Same as ASCII	Same as ASCII	0-1	slash
(	Same as ASCII	Same as ASCII	12-8-5	left parenthesis
)	Same as ASCII	Same as ASCII	11-8-5	right parenthesis
$	Same as ASCII	Same as ASCII	11-8-3	dollar sign
=	Same as ASCII	Same as ASCII	8-6	equals
,	Same as ASCII	Same as ASCII	0-8-3	comma
.	Same as ASCII	Same as ASCII	12-8-3	period
>	Same as ASCII	Same as ASCII	12-8-4	less than
<	Same as ASCII	Same as ASCII	0-8-6	greater than
;	Same as ASCII	Same as ASCII	11-8-6	semicolon
:	:	[1] :	8-2	colon
%	%	[1] %	0-8-4	percent
#	#	≡	8-3	number (CDC - identity)

Note: 1. On CDC systems using a 63-character graphic set, the % character is displayed as a " : ". No separate "%" graphic exists.

Table A-2. EBCDIC Character Set for Use with RJE (Continued)

GRAPHIC			PUNCH	
ASCII	IBM	CDC	CODE (029)	COMMENTS
[	¢	[	12-8-2	left bracket (IBM - cent)
]	!	]	11-8-2	right bracket (IBM - exclamation)
"	"	≠	8-7	quote (CDC not equal)
_	_	→	0-8-5	underscore (CDC - right arrow)
'	'	↑	8-5	apostrophe (CDC - up arrow)
?	?	↓	0-8-7	question (CDC - down arrow)
@	@	≤	8-4	at (CDC - less than or equal)
&	&	∧	12	ampersand (CDC - circumflex)
^	¬	¬	11-8-7	not (ASCII - circumflex)
!	|	∨	12-8-7	exclamation, vertical bar, carat
\	[2]	≥	0-8-2	back slash, greater than-equal

IBM GRAPHIC	PUNCH CODE (029)	COMMENTS
a	12-0-1	
b	12-0-2	
c	12-0-3	
d	12-0-4	
e	12-0-5	
f	12-0-6	
g	12-0-7	
h	12-0-8	
i	12-0-9	
j	12-11-1	
k	12-11-2	
l	12-11-3	
m	12-11-4	
n	12-11-5	
[3] o	12-11-6	
p	12-11-7	
q	12-11-8	
r	12-11-9	
s	11-0-2	
t	11-0-3	
u	11-0-4	
v	11-0-5	
w	11-0-6	
x	11-0-7	
y	11-0-8	
z	11-0-9	

Notes: 2. No graphic exists.
3. While these characters can be transmitted to a host system via the RJE facility, they *cannot* be read into a BASIC program.

HOW TO PREPARE A PAPER TAPE OFF-LINE

APPENDIX B

To prepare a BASIC program on paper tape for input:

1. Set terminal status to "LOCAL."
2. Press the ON button on the paper tape punch.
3. Press control-@ (or the HERE IS key if available) several times to put leading feed holes on the tape.
4. Type the program as usual, **following each line with *X-OFF* (control-S), *return,line feed.***
5. Press control-@ (or HERE IS) several times to put trailing feed holes on the tape.
6. Press the OFF button on the paper tape punch.

The standard on-line editing features, such as line delete and character delete, may be punched on paper tape.

Pressing the BACKSPACE button on the paper tape punch, then the RUBOUT or DEL key on the keyboard, physically deletes the previous character from the paper tape.

An alternate method of producing a punched paper tape copy of a program is to use the PUNCH command. (See the PUNCH command.)

Programs punched onto paper tape in the above manner, or produced by the PUNCH command, may be input to the system through the paper tape reader after typing the TAPE command.

The paper tape punch may also be used off-line to prepare a list of data that will be input in a BASIC program at INPUT, LINPUT, or ENTER statements. Using the same procedure described above for preparing a BASIC program, type each line of data in the following format.

data item, data item, . . . , data item *X-OFF* (*return*) (*line feed*)

Note: When using strings in a line of data, each string must be enclosed in quotes, for example: "AAA", "BBB", 98, 95, "CCC". (For further discussions of data format, refer to the INPUT, LINPUT, and ENTER statements.)

The number of characters permitted in a line of data will vary with system configuration.

To use the data in a BASIC program, insert the data tape in the paper tape reader and set the reader to AUTO. The data will then automatically be read in at INPUT, LINPUT, and ENTER statements when the program is run.

ERROR MESAGES

APPENDIX C

USER COMMAND ERROR MESSAGES

Specific command error messages are listed below along with the commands which generate them. The message ILLEGAL FORMAT is produced by all commands when their parameter string is improperly constructed. The messages listed here usually indicate a syntactically correct command which is rejected for the reason indicated.

APPEND

ENTRY IS A FILE
EXECUTE ONLY
NO COMMON AREA ALLOWED
NO SUCH PROGRAM
PROGRAM TOO LARGE
SEMI-COMPILED PROGRAM
SEQUENCE NUMBER OVERLAP
UNABLE TO RETRIEVE FROM LIBRARY

CREATE

CONFLICTING ACTION BY OTHER USER
DUPLICATE ENTRY
ILLEGAL PARAMETER
LIBRARY SPACE FULL
SYSTEM OVERLOAD
UNSUCCESSFUL, PURGE AND RETRY

CSAVE

DUPLICATE ENTRY
LIBRARY SPACE FULL
NO PROGRAM
NO PROGRAM NAME
OUT OF STORAGE
RUN ONLY
SYSTEM OVERLOAD
UNSUCCESSFUL, TRY AGAIN

(Since execution of this command requires some of the same processing as the RUN command, it may also produce many of the execution errors listed in a separate section.)

DELETE

NOTHING DELETED

EXECUTE

ENTRY IS A FILE
NO SUCH PROGRAM
PROGRAM TOO LARGE
UNABLE TO RETRIEVE FROM LIBRARY

(Since this command includes the processing of a RUN command, it can also produce any of the execution errors listed in a separate section.)

FILE

CONFLICTING ACTION BY OTHER USER
DEVICE UNAVAILABLE
DUPLICATE ENTRY
ILLEGAL PARAMETER
LIBRARY SPACE FULL
RECORD SIZE TOO LARGE
SYSTEM OVERLOAD
UNSUCCESSFUL, PURGE AND RETRY

GET

ENTRY IS A FILE
EXECUTE ONLY
NO SUCH PROGRAM
PROGRAM TOO LARGE
UNABLE TO RETRIEVE FROM LIBRARY

HELLO

ILLEGAL ACCESS
NO TIME LEFT

LIST

CHARACTERS AFTER COMMAND END
PROGRAM BAD
RUN ONLY

LOCK

NO SUCH ENTRY

MESSAGE

CONSOLE BUSY

MWA

ASCII FILE, NOT ALLOWED
ENTRY IS A PROGRAM
INSUFFICIENT CAPABILITY
NO SUCH ENTRY
NO SUCH ID

NAME

ONLY 6 CHARACTERS ACCEPTED

PRIVATE

NO SUCH ENTRY

PROTECT

NO SUCH ENTRY

PUNCH

ASCII FILE NOT PERMITTED
ASCII FILE REQUIRED
RUN ONLY
PROGRAM BAD
CHARACTERS AFTER COMMAND END

PURGE

FILE IN USE
NO SUCH ENTRY

RENUMBER

BAD PARAMETER
SEQUENCE NUMBER OVERFLOW/OVERLAP

RUN

(See the list of execution errors in another section.)

SAVE

DUPLICATE ENTRY
LIBRARY SPACE FULL
NO PROGRAM
NO PROGRAM NAME
OUT OF STORAGE
RUN ONLY
SYSTEM OVERLOAD
UNSUCCESSFUL, TRY AGAIN

SWA

ENTRY IS A PROGRAM
NO SUCH ENTRY

UNRESTRICT

NO SUCH ENTRY

Use of the *OUT=file name* construct in one of the above commands can also produce the following messages

DEVICE UNAVAILABLE
FILE/DEVICE BUSY OR NOT PRESENT
FILE READ ONLY OR NOT ASCII
RECORD SIZE TOO LARGE
END OF FILE

If users other than account A000 attempt to use certain commands reserved for the system manager they will receive the message

PRIVILEGED COMMAND

If a program entered under control of the TAPE command contained errors, execution of the next command entered will be replaced by printing of the error messages followed by

LAST INPUT IGNORED, RETYPE IT

LANGUAGE PROCESSOR ERROR MESSAGES

The following messages are output by the BASIC language processor to indicate errors or possible errors in users' BASIC programs.

SYNTAX ERRORS

One of the following error messages will be produced after entry of a BASIC statement with incorrect syntax. In all cases but the last the line will be rejected rather than added to the current program.

OUT OF STORAGE
ILLEGAL OR MISSING INTEGER
EXTRANEOUS LIST DELIMITER
MISSING ASSIGNMENT OPERATOR
CHARACTERS AFTER STATEMENT END
MISSING OR ILLEGAL SUBSCRIPT
MISSING OR BAD LIST DELIMITER
MISSING OR BAD FUNCTION NAME
MISSING OR BAD SIMPLE VARIABLE
MISSING OR ILLEGAL 'OF'
MISSING OR ILLEGAL 'THEN'
MISSING OR ILLEGAL 'TO'
MISSING OR ILLEGAL 'STEP'
MISSING OR ILLEGAL DATA ITEM
ILLEGAL EXPONENT
SIGN WITHOUT NUMBER
MISSING RELATIONAL OPERATOR
ILLEGAL READ VARIABLE
ILLEGAL SYMBOL FOLLOWS 'MAT'
MATRIX CANNOT BE ON BOTH SIDES
NO '*' AFTER RIGHT PARENTHESIS
NO LEGAL BINARY OPERATOR FOUND
MISSING LEFT PARENTHESIS
MISSING RIGHT PARENTHESIS
PARAMETER NOT STRING VARIABLE
UNDECIPHERABLE OPERAND
MISSING OR BAD ARRAY VARIABLE
STRING VARIABLE NOT LEGAL HERE
MISSING OR BAD STRING OPERAND
NO CLOSING QUOTE
255 CHARACTERS MAX FOR STRING
STATEMENT HAS EXCESSIVE LENGTH
MISSING OR BAD FILE REFERENCE
'PRINT' MUST PRECEDE 'USING'
ILLEGAL OPERAND AFTER 'USING'
VARIABLE MISSING OR WRONG TYPE
OVER/UNDERFLOWS - WARNING ONLY

EXECUTION ERRORS

Erroneous conditions discovered during execution of a program (i.e., following entry of a RUN or EXECUTE command) will result in one of the messages below. In addition such errors cause termination of execution.

UNDEFINED STATEMENT REFERENCE
NEXT WITHOUT MATCHING FOR
SAME FOR-VARIABLE NESTED
FUNCTION DEFINED TWICE
VARIABLE DIMENSIONED TWICE
LAST STATEMENT NOT 'END'
UNMATCHED FOR
UNDEFINED FUNCTION
ARRAY TOO LARGE
ARRAY OF UNKNOWN DIMENSIONS
OUT OF STORAGE
DIMENSIONS NOT COMPATIBLE
CHARACTERS AFTER COMMAND END
BAD FORMAT OR ILLEGAL NAME
MISSING OR PROTECTED FILE
GOSUBS NESTED TWENTY DEEP
RETURN WITH NO PRIOR GOSUB
SUBSCRIPT OUT OF BOUNDS
NEGATIVE STRING LENGTH
NON-CONTIGUOUS STRING CREATED
STRING OVERFLOW
OUT OF DATA
DATA OF WRONG TYPE
UNDEFINED VALUE ACCESSED
MATRIX NOT SQUARE
REDIMENSIONED ARRAY TOO LARGE
NEARLY SINGULAR MATRIX
LOG OF NEGATIVE ARGUMENT
SQR OF NEGATIVE ARGUMENT
ZERO TO ZERO POWER
NEGATIVE NUMBER TO REAL POWER
ARGUMENT OF SIN OR TAN TOO BIG
TOO MANY FILES STATEMENTS
NON-EXISTENT FILE REQUESTED
WRITE TRIED ON READ-ONLY FILE
END-OF-FILE/END OF RECORD
STATEMENT NOT IMAGE
NON-EXISTENT PROGRAM REQUESTED
CHAIN REQUEST IS A FILE
PROGRAM CHAINED IS TOO LARGE
COM STATEMENT OUT OF ORDER
ARGUMENT OUT OF RANGE

BAD FORMAT STRING SUBSCRIPT
BAD FILE READ
BAD FILE WRITE DETECTED
CAN'T READ PROGRAM CHAINED TO
NO ACCESS ALLOWED
PROGRAM BAD
STATEMENT NUMBER OUT OF BOUNDS
NO ACCESS AT THIS TIME
ASCII FILE NOT PERMITTED
RETURN VARIABLE NEEDED
ASCII FILE REQUIRED
MISSING OR INVALID COMMAND
READ TRIED ON WRITE-ONLY FILE
MISSING FORMAT SPECIFICATION
ILLEGAL OR MISSING DELIMITER
NO CLOSING QUOTE
BAD CHARACTER AFTER REPLICATOR
REPLICATOR ZERO OR TOO LARGE
MULTIPLE SIGNS
MULTIPLE DECIMAL POINTS
BAD FLOATING SPECIFICATION
ILLEGAL CHARACTER IN FORMAT
ILLEGAL FORMAT FOR STRING
MISSING RIGHT PARENTHESIS
MISSING REPLICATOR
TOO MANY PARENTHESIS LEVELS
MISSING LEFT PARENTHESIS
ILLEGAL FORMAT FOR NUMBER

EXECUTION WARNINGS

The following messages occur under the same circumstances as the execution errors above. However, they represent recoverable or questionable but allowable conditions and therefore do not terminate execution.

BAD INPUT, RETYPE FROM ITEM xxxx
LOG OF ZERO - WARNING ONLY
ZERO TO NEGATIVE POWER-WARNING
DIVIDE BY ZERO - WARNING ONLY
EXP OVERFLOW - WARNING ONLY
OVERFLOW - WARNING ONLY
UNDERFLOW - WARNING ONLY
EXTRA INPUT - WARNING ONLY
TRANSMISSION ERROR. REENTER
OVER/UNDERFLOWS - WARNING ONLY

TERMINAL INTERFACE

APPENDIX D

User terminals can be operated in either of two modes, on-line or off-line. In on-line mode, connection to the computer is established, a log-on procedure is performed, and the user is in contact with the computer through the 2000 Access System. This system accepts and executes any legal command entered by the user. Illegal commands are rejected, usually with an informative message printed or displayed on the terminal.

To enter a command, type either the short or full form of the command; if additional parameters are required or permitted, type a hyphen, then the parameters. Terminate the command by pressing (*return*). Some commands cause an obvious response from the system such as a listing or punching operation. Other commands result in computer operations where the only response is the generation of a (*line feed*), at your terminal indicating that the system has accepted the command and is ready for another entry.

Terminals with paper tape punching capabilities may be used to prepare paper tape in off-line mode.

Ten types of user terminals can be connected to the 2000 Access System. Nine generate ASCII code and one generates CALL 360 or PTTC/EBDC (non-ASCII) code.

The following user terminals generate ASCII code:

- HP 2640A Interactive Display Terminal
- HP 2749A Teleprinter Terminal
- General Electric TermiNet 300 Data Communications Terminal, Model B (10/15/30 cps transfer rates) with Paper Tape Reader/Punch, Option 2 or HP 2762 Terminal

  Note: The terminal must be strapped for "ECHO-PLEX".

- General Electric TermiNet 1200
- Memorex 1240 Communications Terminal (10/15/30 cps transfer rates)
- Execuport 300 Data Communications Transceiver Terminal
- ASR-37 Teleprinter Terminal with Paper Tape Reader/Punch

  Note: If the terminal is equipped with the Shift Out (SO) feature, SO must be disabled because the 2000 Access System does not allow use of this feature.

- Texas Instruments Silent 700

The following user terminal generates non-ASCII code:

- IBM 2741 Communication Terminal

Note: The terminal must be connected to the system over telephone lines. In addition, the terminal must be equipped with the following features:

1. Interrupt, Receive (IBM #4708) and Transmit (IBM #7900) associated with the terminal's ATTN key.
2. Dial-Up (IBM #3255) to enable system connection through a 103A modem or acoustic coupler.

Any terminal equipped with the automatic(*line feed*)feature (operator selectable) must be operated with this feature OFF.

Note: Although cursor, form feed, horizontal and vertical tabulation, and various special function keys are provided on specific types of user terminals, these capabilities are not supported by the 2000 Access System. Some of these operations may be requested from the keyboard, but results are unpredictable. Features provided by 2000 Access BASIC, such as the TAB, SPA, and LIN functions, and the PRINT and PRINT USING statements, should be used to control output format. However, terminals equipped with automatic line feed after carriage return or on end of line may cause unpredictable results. These functions and statements are described in other sections of this manual.

IBM 2741 COMMUNICATIONS TERMINAL INTERFACE

Because the IBM 2741 terminal generates non-ASCII code, special consideration must be given to the representation of several ASCII characters and functions which are not available in the 2741 character set.

For input from a 2741 terminal, these characters (and some of the functions) are simulated by entry of a two-character code. The first character of this code is the cent symbol (¢). The cent symbol is followed by one of several alphanumeric or special characters to compose a unique code representing one ASCII character or function.

On input from a 2741 terminal, the two-character code is translated into the internal ASCII code. On output to a 2741 terminal, ASCII code is translated into the appropriate two-character representation.

The IBM 2741 Communications Terminal must be equipped with the interrupt feature associated with the *ATTN* key. This key represents the *break* function; it is used to terminate program or command execution.

Any CALL/360 or PTTC/EBCD characters that do not have an equivalent ASCII character are ignored on input.

Table D-1 shows 2741 terminal representation of ASCII characters and functions.

Table D-1. IBM 2741 ASCII Character Simulation

ASCII Graphic Control Character	IBM 2741 Character Representation		User Terminal Function	IBM 2741 Character Representation	
	CALL/360	PTTC/EBCD		CALL/360	PTTC/EBCD
[	¢(	¢(	*control* ①	¢C	¢C
\	¢/	¢/	*break*	ATTN key	ATTN key
]	¢)	¢)			
^	↑	¢A			
`	¢'	¢'			
{	¢0	¢0			
}	¢S	¢S			
~	¢T	¢T			
—	—	—			
ESC	¢E	¢E			
FS	¢F	¢F			
GS	¢G	¢G			
RS	¢R	¢R			
US	¢U	¢U			

① Code must be followed by an appropriate alphabetic character; otherwise, it is ignored.

Examples:

Action	Code Required
Input line deletion (control X)	*¢CX* (See Note)
Character deletion (backspace) (no character will be echoed to confirm the deletion)	*¢CH*

Note: This entry must be followed by return. Otherwise, it is ignored.

ADDITIONAL LIBRARY FEATURES

APPENDIX E

The system operator has several program and file movement capabilities of which the user should be aware. In addition, there are some commands available to the operator at system startup time which allow storage and retrieval of programs and files on magnetic tape. These operator commands, and their functions, are listed here. The discussions of operator commands that follow assume that you are familiar with the library and security structure of the system (refer to Section VIII).

BESTOW

This command enables the operator to transfer a program or file or an entire library from one account to another. Individual library entries may be transferred any time the system is running. Entire libraries may only be transferred when no users are logged on and the system is running. Library entries which are PRIVATE or LOCKED will not be transferred. UNRESTRICTED or PROTECTED entries remain UNRESTRICTED or PROTECTED. An MWA, (Multiple Write Access) file remains MWA only if the new library's idcode has the MWA capability.

COPY

This command is used to make a duplicate copy of any user program or file in the library of any other user (or the same user). The copy may be given a new name at the time it is created. A program or a file of fewer than 200 blocks may be transferred any time the system is running. Files of greater than 200 blocks may be copied only when no users are logged on and the system is running. Entries which are PRIVATE or LOCKED will not be transferred. UNRESTRICTED or PROTECTED entries remain UNRESTRICTED or PROTECTED. An MWA (Multiple Write Access) file remains MWA only if the new library's idcode has the MWA capability.

LOAD

At system startup time, the system operator may use the LOAD command to load selected programs and files, entire user libraries, or all entries from magnetic tape. This tape must have been produced by a DUMP, SLEEP, or HIBERNATE. Entries already on the system will not be loaded. Program and file states (UNRESTRICTED, LOCKED, PRIVATE, or PROTECTED) will not be altered. An MWA (Multiple Write Access) file will remain MWA only if the library into which it is loaded has the MWA capability.

RESTORE

At system startup time, the system operator may use the RESTORE command to load selected programs and files, entire user libraries, or all entries from magnetic tape. This tape must have been produced by a DUMP, SLEEP, or HIBERNATE. The difference between LOAD and RESTORE is that LOAD does not replace existing entries; RESTORE does. Program and file states (UNRESTRICTED, PROTECTED, LOCKED, or PRIVATE) will not be altered. An MWA (Multiple Write Access) file will remain MWA only if the library into which it is loaded has the MWA capability.

DUMP

The DUMP command can be used by the system operator at system startup time to write selected user programs and files, entire user libraries, or all entries on a system to magnetic tape. Dump tapes are useful for archival storage, backup, or transferring entries between Access systems. The security and access states of programs and files are not altered when they are dumped.

FORMAL SYNTAX FOR 2000/ACCESS BASIC

APPENDIX F

This appendix contains a precise definition of the 2000/Access BASIC language. The descriptions listed here can be used to clarify the less formal definitions used in other parts of the manual. The syntactical grammar is described in a formal metalanguage derived from the Backus-Naur Form (BNF) of syntax definition.

The BNF notation consists of "productions" or syntax equations, each of which is in the following form:

<syntactic entity> ::=<syntactic expression>

This can be read as "the entity on the left is composed of the ordered collection of one or more of the expressions on the right.

- If the entity has more than one expression, they are separated by a vertical bar "|". These expressions represent choices for any given expansion of the entity.

- Square brackets "[ ]" are used to enclose optional portions of expressions. The brackets can be nested (i.e., options can have options), and alternative options are expressed as a list of expressions separated by vertical bars, all enclosed within the brackets.

- Expressions will normally contain one or more entities. These can be expanded by substituting the right-hand side of the definitions for the entities.

- The syntax equations may be recursive (the entity on the left may appear in an expression used to define it. When this occurs there is always at least one alternative which does not define the entity in terms of itself. This allows definitions where there are multiple occurrences of the same component.

- In some definitions the right-hand side of the equation will be a textual description rather than an expression.

- Numerals in definitions refer to notes which follow the formal definitions.

<letter> : := A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<digit> : := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<signop> : := + | -

<relational operator> : := < | <= | = | >= | > | <> | #

<integer> : := <digit> | <integer><digit>

<character string> : := < character> | <character string><character>

<character> : := Any ASCII character[1]

<constant> : := [<signop>]<numeric constants> | <literal string>

<numeric constant> : := <significant part> [<exponent>]

<significant part> : := <integer> | [<integer>].[<integer>]

<exponent> : := E [<signop>]<integer> [2]

<literal string> : := <quoted string> | '<integer> [3] [<quoted string>]| <literal string> '<integer>[3] [<quoted string>]

<quoted string> : := " [<character string>] " [4]

<numeric variable> : := <numeric simple variable> | <subscripted variable>

<numeric simple variable> : := <letter> [<digit>]

<subscripted variable> : := <array id> (<subscript> [,<subscript>]) [17]

<array id> : := <letter>

<subscript> : := <numeric expression >

<return variable> : := <numeric variable>

<string variable> : := <string simple variable> [(<first character> [, <last character>])] [17]

<string simple variable> : := <letter> [0 | 1] $

<first character> : := <numeric expression>

<last character> : := <numeric expression>

<numeric expression> : := <conjunction> | <numeric expression> OR <conjunction>

<conjunction> : := <relation> | <conjunction> AND <relation>

<relation> : := <minmax> |
<relation> <relational operator> <minmax>

<minmax> : := <sum> | <minmax> MIN <sum> |
<minmax> MAX <sum>

<sum> : := <term> | <sum> <signop> <term>

<term> : := <subterm> | <term> * <subterm> |
<term>/<subterm>

<subterm> : := [<signop>] <factor> | NOT <factor>

<factor> : :=<primary>|<factor>↑[18]<primary>|
<factor> ** <primary>

<primary> : := <numeric constants> | <numeric variable> | <parameter> [5] |
<functional> | (<numeric expression>)

<parameter> : := <letter> [<digit>]

<functional> : := <function id> (<numeric expression>) |
<pre-defined function> (<numeric expression>) |
LEN(<source string>) | NUM(<source string>) |
POS(<source string> , <source string>)

<function id> : := FN <letter>

<pre-defined function> : := ABS|ATN|BRK|COS|EXP|INT|ITM|LOG|REC|RND|SGN|
SIN|SQR|TAN|TIM|TYP

<string expression> : := <source string> | CHR$(<numeric expression>) |
UPS$(<source string>)

<source string> : := <literal string> | <string variable>

<file reference> : := <file expression> [, <record expression>]

<file expression> : := # <numeric expression>

<record expression> : := <numeric expression>

<program> : := <program statement> |
<program> <program statement> [6]

<program statement> : := <statement number> <BASIC statement>

<statement number> : := <integer> [7]

<BASIC statement> : := <LET statement> | <IF statement> |

<trap statement> | <GOTO statement> |

<GOSUB statement> | <RETURN statement> |

<FOR statement< | <NEXT statement> |

<STOP statement> | <END statement> |

<DATA statement> | <READ statement> |

<BASIC statement> (Continued)	<INPUT statement> | <ENTER statement> |
	<LINPUT statement> | <RESTORE statement> |
	<PRINT statement> | <PRINT USING statement> |
	<IMAGE statement> | <REM statement> |
	<DIM statement> | <COM statement> |
	<DEF statement> | <FILES statement> |
	<ASSIGN statement> | <CHAIN statement> |
	<CONVERT statement> | <MAT statement> |
	<CREATE statement> | <PURGE statement> |
	<ADVANCE statement> | >UPDATE statement> |
	<LOCK statement> | <UNLOCK statement> |
	<SYSTEM statement>
<LET statement>	: := [LET] <left part> <numeric expression> |
	[LET] <destination string> = <string expression>
<left part>	: := <numeric variable> = | <left part> <numeric variable> =
<destination string>	: := <string variable>
<IF statement>	: := IF <decision expression> THEN <statement number>
<decision expression>	: := <numeric expression> | <string variable>
	<relational operator> <string expression>
<trap statement>	: := IF END <file expression> THEN <statement number>
<GOTO statement>	: := GOTO <statement number> |
	GOTO <numeric expression> OF <statement number list>
<statement number list>	: := <statement number> |
	<statement number list> , <statement number>
<GOSUB statement>	: := GOSUB <statement number> |
	GOSUB <numeric expression> OF <statement number list>
<RETURN statement>	: := RETURN
<FOR statement>	: := FOR <for variable> = <initial value> TO
	<final value> [STEP <step size>]
<for variable>	: := <numeric simple variable>
<initial value>	: := <numeric expression>

<final value> : := <numeric expression>

<step size> : := <numeric expression>

<NEXT statement> : := NEXT <for variable>

<STOP statement> : := STOP

<END statement> : := END

<DATA statement> : := DATA <constant list>

<constant list> : := <constant> | <constant list> , <constant>

<READ statement> : := READ [<file reference> ;] <read variable list> |
READ <file reference>

<read variable list> : := <read variable> |
<read variable list> , <read variable>

<read variable> : := <numeric variable> | <destination string>

<INPUT statement> : := INPUT <read variable list>

<ENTER statement> : := ENTER # <numeric variable> | ENTER [# <numeric variable> ,]
<numeric expression> , <return variable> ,
<read variable>

<LINPUT statement> : := LINPUT [<file expression> ;] <string variable>

<RESTORE statement> : := RESTORE [<statement number>]

<PRINT statement> : := <type statement> | <file write statement>

<type statement> : := PRINT [<print list> [, | ;]]

<print list> : := <print expression> |
<print list> , <print expression> | [8]
<print list> ; <print expression>

<print expression> : := <numeric expression> | <string expression> |
<print function>

<print function> : := <print function id> (<numeric expression>)

<print function id> : := LIN|SPA|TAB|CTL [9]

<file write statement> : := PRINT <file reference> [;<write list> | ; END]

<write list> : := <print list> [, [END] | ; [END]]

<PRINT USING statement> : := PRINT USING <format part> [; <using list>]

<format part> : := <statement number> | <string variable> |
"<format string>"

<format string> : := [[<carriage control> ,] <format list>] [10]

<carriage control> : := + | - | #

<format list> : := <format list element> | / |
<replicator> (<format list>) | [11]
<format list> , <format list element> |
<format list> / [<format list element>]

<format list element> : := <literal string> [12] | <X list> |
<string specification> |
<integer specification> |
<fixed specification> |
<floating specification>

<X list> : := <X part> | <X list> <X part>

<X part> : := [<replicator>] X

<replicator> : := <integer> [13]

<string specification> : := [<X list>] <A part> | <string specification> <A part> |
<string specification> <X list>

<A part> : := [<replicator>] A

<integer specification> : := [S] <num spec> [14]

<num spec> : := [<X list>] <D part> | <num spec> <D part> |
<num spec> <X list>

<D part> : := [<replicator>] D

<fixed specification> : := [S] <num spec> . [<num spec>] | [14]
[S] . <num spec>

<floating specification> : := <integer specification> E [<X list>] |
<fixed specification> E [<X list>]

<using list> : := <using expression> |
<using list> , <using expression>

<using expression> : := <numeric expression> | <string variable> |
<print function>

<IMAGE statement> : := IMAGE <format string>

<REM statement> : := REM [<character string>]

<DIM statement> : := DIM <dimspec> | <DIM statement> ,<dimspec>

<dimspec> : := <array id> (<bound> [, <bound>])
<string simple variable> (<bound>) [17]

<bound> : := <integer> [15]

<COM statement> : := COM <com list element> |
<COM statement> , <com list element>

<com list element> : := <dimspec> | <numeric simple variable> |
<string simple variable>

<DEF statement> : := DEF <function id> (<parameter>) =
<numeric expression>

<FILES statement> : := FILES <file name designator> | [10]
<FILES statement> , <file name designator>

<file name designator> : := [$|*] <name> [. <account id>] | *

<name> : := A string of 1 to 6 letters and/or digits

<account id> : := <letter> <digit> <digit> <digit>

<ASSIGN statement> : := ASSIGN <file name> , <file number> ,
<return variable> [, <protection mask>]
[, <restriction>] |
ASSIGN * , <file number> [, <return variable>]

<file name> : := <source string>

<file number> : := <numeric expression>

<protection mask> : := <source string>

<restriction> : := RR|NR|WR

<CHAIN statement> : := CHAIN [<return variable>,] <program name>
[,<numeric expression>]

<program name> : := <source string>

<CONVERT statement> : := CONVERT <numeric expression> TO <string variable> |
CONVERT <source string> TO <numeric variable>
[, <statement number>]

<MAT statement> : := <MAT READ statement> | <MAT INPUT statement> |
<MAT PRINT statement> |
<MAT PRINT USING statement> |
<MAT initialization statement> |
<MAT assignment statement>

<MAT READ statement> : := MAT READ [<file reference>;] <actual array> |
<MAT READ statement> , <actual array>

<actual array> : := <array id> [<dimensions>]

<dimensions> : := (<numeric expression> [, <numeric expression>])

<MAT INPUT statement> : := MAT INPUT <actual array> |
<MAT INPUT statement> , <actual array>

<MAT PRINT statement> : := <mat type statement> | <mat file write statement>

<mat type statement> : := MAT PRINT <mat print list> [, | ;]

<mat print list> : := <mat print expression> |
<mat print list> , <mat print expression> |
<mat print list> ; <mat print expression>

<mat print expression> : := <array id> | <print function>

<mat file write statement> : := MAT PRINT <file reference> ; <mat print list>
[, [END] | ; [END]] |
MAT PRINT <file reference> ; END

<MAT PRINT USING statement> : := MAT PRINT USING <format part> [; <mat print list>]

<MAT initialization statement> : := MAT <array id> = <initialization function>
[<dimensions>]

<initialization function> : := ZER|CON|IDN

<MAT assignment statement> : := MAT <array id> [<signop> <array id>] |
MAT <array id> = <array id> * <array id> | [16]
MAT <array id> = TRN (<array id>) | [16]
MAT <array id> = INV (<array id>) |
MAT <array id> = (<numeric expression>) * <array id>

<CREATE statement> : := CREATE <return variable> , <file name> ,
<file length> [, <record size>]

<file length> : := <numeric expression>

<record size> : := <numeric expression>

<PURGE statement> : := PURGE <return variable> , <file name>

<ADVANCE statement> : := ADVANCE <file expression> ; <skip count> ,
<return variable>

<skip count> : := <numeric expression>

<UPDATE statement> : := UPDATE <file expression> ; <numeric expression> |
UPDATE <file expression> ; <source string>

<LOCK statement> : := LOCK <file expression> [, <return variable>]

<UNLOCK statement> : := UNLOCK <file expression> [, <return variable>]

<SYSTEM statement> : := SYSTEM <return variable> , <source string> | [9]
SYSTEM <string variable> , <source string>

NOTES:

1. The following ASCII characters are stripped by the system from terminal input and therefore cannot be entered directly: null, control-H, line-feed, carriage-return, X-OFF, control-X, and rubout.

2. Exponent integers are limited to exactly one or two digits.

3. The value of an integer used to supply a character within a literal string must lie between 0 and 255 inclusive.

4. The double quote character ('') cannot appear within a quoted string.

5. A parameter primary can appear only in the defining expression of a DEF statement.

6. The last statement of a program must be an END statement.

7. A sequence number must lie between 1 and 9999 inclusive.

8. Print expressions which are literal strings need not be separated from preceding or following print expressions by semicolons or commas.

9. The CTL function and SYSTEM statement are requests for operating system services rather than a part of the BASIC language proper. They are included here as a convenient reference for their syntax. Users are explicitly cautioned that these constructs are not portable to any other Hewlett-Packard implementation of the BASIC language.

10. Any character string is accepted by the language grammar for a format string or a list of file names. The syntax of the string is checked when it is used during execution.

11. Groups in format lists can be nested only two levels deep.

12. In order to embed a literal string as a format list element into the quoted form of a format string (which itself is a literal string within the language grammar), the delimiting double quote characters of the literal string must be represented by means of the apostrophe convention (i.e., '34). The apostrophe convention is not needed or recognized when the format string occurs within an IMAGE statement or as the contents of a string variable. In no case can a double quote appear as a character within a literal string embedded in the format string referenced by a PRINT USING statement.

13. A replicator must lie between 1 and 255 inclusive.

14. An S can appear before, after, or between any two parts of a num spec except immediately following a replicator. Only one S can appear within one integer specification, fixed specification, or floating specification.

15. An array bound must lie between 1 and 9999 inclusive; a string variable bound must lie between 1 and 255 inclusive.

16. An array cannot be transposed into itself nor can it be both an operand and the result of a matrix multiplication.

17. Parentheses, ( ), and square brackets, [ ], are accepted interchangeably by the BASIC language.

18. A circumflex (^) may replace the up arrow (↑) on some terminals.

INDEX

READER COMMENT SHEET

22687-90001 SEP 1975

HP 2000/Access BASIC
Reference Manual

We welcome your evaluation of this manual. Your comments and suggestions help us improve our publications. Please use additional pages if necessary.

Is this manual technically accurate?

Is this manual complete?

Is this manual easy to read and use?

Other comments?

FROM:

Name _____

Company _____

Address _____

_____

_____

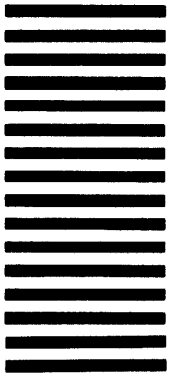
FOLD FOLD

FIRST CLASS
PERMIT NO. 141
CUPERTINO
CALIFORNIA

BUSINESS REPLY MAIL

No Postage Necessary if Mailed in the United States Postage will be paid by

Manager Customer Engineering
Hewlett-Packard Company
Data Systems Division
11000 Wolfe Road
Cupertino, California 95014



FOLD FOLD

HEWLETT hp PACKARD

Sales and service from 172 offices in 65 countries.
11000 Wolfe Road, Cupertino, California 95014

Printed in U.S.A. 9/75

PART NO. 22687-90001