MODEL 4000

Interactive Timesharing System

USER'S MANUAL

Basic Timesharing Inc.

LEADERS IN THE MANUFACTURE AND SUPPORT OF TIMESHARED COMPUTER SYSTEMS

FOREWORD

THE BTI 4000 SYSTEM

Basic Timesharing 4000 series computer hardware and software are designed specifically as an integrated timesharing system, and all system computing resources are shared among all active users on a round-robin basis.

Access to the computer is done through an account identifier with an associated, non-printing password. Up to 5800 user accounts, selected from 26,000 possible account identifiers, can be assigned at one time on the BTI 4000. The BTI hierarchy of supervisory and user accounts and their associated libraries permits easy allocation of computer resources among organizational departments, projects, etc. Any combination of software application packages, application users and traditional timesharing users can co-exist on the 4000 system.

Proprietary software packages may be installed and maintained on BTI 4000 system which are not under the direct management of the software package owner. Reciprocal security mechanisms permit the software owner to access his package, even through a remote terminal telephone connection, while avoiding the risk of security breaches, either in the proprietary software, or in the host computer.

SYSTEM MANAGEMENT

The BTI 4000 system is designed for simplicity of management. This User's Manual, the System Manager's Manual and a set of Public Library Program descriptions provide all the instruction necessary for the system manager to make full use of the computer. Since all system manager control and reporting functions can be implemented through any system port, there is no need for a system operator's console.

BASIC-X LANGUAGE

BASIC-X, the BTI 4000 user language, is an extended version of the popular, easy to use BASIC language. BASIC-X has been continually improved since its introduction in 1970, and includes matrix, logical and string operators, and string arithmetic with up to 252 digits of decimal precision. Programs may be shared among many users. The BASIC-X file subsystem permits dynamic file creation and deletion from a program; variable length, multi-record file buffers for fast data access; up to 63 concurrently linked files; linked files declared in COMMON, eliminating the need to re-link files when chaining programs; non-interfering file sharing to prevent update errors. Command files can provide the user with virtual "one-step" man/machine interface. Features to facilitate program development include line-by-line format and syntax verification; automatic FOR . . . NEXT loop indentation and Execute Immediate Mode for dynamic program debugging.

HOW TO USE THIS MANUAL

This manual is intended primarily as a reference for current and prospective users of BTI interactive timesharing systems. The sample programs may be duplicated at the user's own terminal, and beginning users are encouraged to try the samples themselves. The syntax examples are generalized versions of how the statement of command under discussion is to be constructed. Statement and command key verbs are capitalized and their arguments and parameters are italicized. Often a two-word argument descriptor is spelled as one word, e.g., "line number" appears as "linenumber". This is done to avoid the implication that the argument may in some way demand a two-part construction.

The text following the word FEATURES throughout this manual gives pertinent information about the syntax such as which arguments are optional, their default values if any, and their specific purposes.

The reader's comments and suggestions are invited and may be addressed to:

Basic Timesharing, Inc.
870 W. Maude Avenue
Sunnyvale, California 94086

Attn: User's Manual

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

Chapter 1

GETTING STARTED

1.1 LOG-ON PROCEDURES

1.1.1 Account Numbers and Passwords

A prescribed log-on procedure must be followed in order for a prospective user to be recognized by the system. The user must have a valid account number and password combination to gain access to the system.

Account numbers may be opened by the System Manager, or a Master Account may open accounts within the same letter block. Account numbers consist of one alpha character followed by three numeric digits.

Passwords may be any combination of ten or fewer allowed characters, printing or non-printing. Non-printing or control characters used as passwords provide good security against unauthorized use of an account number, or access to programs, files and data bases. However, the control characters listed below are not recommended for use in non-printing passwords:

G	sounds a bell	J	generates a linefeed
L	initiates a form feed	M	generates a carriage return
R	turns on the paper punch	S	generates an X-OFF and is ignored
D	disconnects some modems	K	generates a vertical tab
I	causes a horizontal tab	P	turns off the cursor on CRTs

Certain account numbers are privileged accounts, with unique system commands or other features. Thus, for example,

D200	the Group Library shared by all D2nn accounts, and referenced by a single $ symbol, e.g., GET-$PROGRAM
D001	the Master Library shared by all Dnnn accounts, and referenced by two $$ symbols, e.g., GET-$$PROGRAM
D002	the Master Account for all Dnnn accounts and not accessible to other users.
@001	the System Library shared by all accounts, and referenced by three $ symbols, e.g., GET-$$$PROGRAM

1.1.2 Log-On Procedures Using a Teletypewriter (or equivalent)

After establishing a connection with the system, either by telephone line or by direct ("hard wire") line, the user may verify that the connection exists by pressing the ESCAPE key. If there is a connection to the computer, then a backslash (\) will be printed. If no connection exists, then there will be no response to the ESCAPE key. The user must identify himself to the computer using the following format and a carriage return:

HEL-*account,password* (carriage return) e.g.,
HEL-X234,PASSWORDOK (carriage return)

Following a valid log-on, the system will respond by executing the system 'HELLO' program. Like any abortable program, the HELLO program and any program to which it is chained can either be run to completion or aborted by striking the BREAK key twice with about a 1 second pause between strikes. In some cases the System Manager or a Master Account may elect to make the HELLO and MESSAGE programs non-abortable. Once the HELLO program and any program to which it is chained is either completed or aborted, the user's swap track is cleared for entry of program statements or execution of programs or of direct commands. HELLO programs or programs to which they chain cannot be listed or punched by the user.

If the account/password combination is not recognized, the system will respond:

INVALID PASSWORD

and the user will be required to issue the log-on command again. About one minute is allowed for the user to accomplish the log-on. After that time if there is no successful log-on, the system will disconnect the line and another call will have to be placed to the computer.

1.1.3 Log-On Procedures Using Other Terminals

While the computer can communicate with either full or half duplex terminals, full duplex (echo-plex, actually) is the preferred mode. This mode permits the user to detect any transmission errors in input characters by examining the echoed characters at the teleprinter. If a given terminal operates only in half duplex, then garbage will be echoed at the terminal in response to the HELLO command. To check for a successful log-on and to modify the computer's transmission mode, the command,

ECHO-OFF

will suppress the computer's echo of input. The command

ECHO-ON

returns the computer to echo-plex transmission.

The system usually operates at 110 baud (10 characters per second) unless certain ports have been preset by the System Manager to expect a different baud rate. Regardless of the preset rate, the baud rate on a port may be modified by the user via the RATe command. For example, after a successful log-on to a 110 baud port, the user may wish to change the transmission rate to 300 baud. This is done by typing

RAT-300

Standard baud rates are 110 with 11-bit characters, or 150, 220, 300, 440, 600, 880, 1200 and 2400, all with 10-bit characters. If any non-standard baud rate between 100 and 2500 or if a non-standard bit pattern is desired, then the user may force the rate and bit pattern by using the RATe command plus one ! symbol if 10-bit characters are desired, or two ! symbols if 11-bit characters are desired. For example,

RAT-1100!!

forces the system to transmit and receive data at 100 characters per second, with 11-bit characters, and

RAT-750!

forces the system to transmit and receive data at 75 characters per second with 10-bit characters.

The standard carriage width at log-on is 72 characters, but the user may modify this downward to as narrow as 20 characters per line, or upward to as wide as 254 characters per line. This modification is done by the CARriage command, thus:

CAR-*n* e.g., CAR-132

would set the carriage width to 132 characters. There is a special form of the CAR command,

CAR-0

which sets the carriage width to "unlimited". This feature is used for plotting on terminals, for CRT cursor control, for backspacing and various other special applications.

Terminals other than a standard teleprinter may require extra time for the carriage to return to the left side of the page following a line of input or output. The following commands are provided to allow for the timing variations found in various terminals. For slower carriage return mechanisms, the time allowed may be altered to fit the current need by the CRD (Carriage Return Delay) command, thus:

CRD-*n* e.g., CRD-20

where n = character times. If the port is operating at 110 baud, (10 cps) then a 2-second delay will result from the command CRD-20. The upper limit on CRD is 100, and the default setting is 0.

The default setting for Line Feed Delay is 0. This may be modified by the LFD command, thus:

LFD-*n* e.g., LFD-30

where n = character times. If the port is operating at 110 baud, (10 cps) then a 3-second delay will result from the command LFD-30. The upper limit on LFD is 100.

The Form Feed Delay may be modified by the FFD command, thus:

FFD-*n* e.g., FFD-200

where n = character times. If the port is operating at 110 baud, (10 cps) then a 20-second delay will result from the command FFD-200. The upper limit on FFD is 255, and the default setting is 10.

1.2 LOG-OFF PROCEDURES

There are four ways for the user to terminate operation from a given account number on a given port.

1. The recommended way to log off the system is to type the command BYE. The system will print out the amount of time used in hours, advance the paper for removal of output, and automatically disconnect the terminal in an orderly manner. The current program in the user's work area is lost.
2. The recommended way to transfer between account numbers is to execute a new HELLO command for the new account number. If a valid HELLO is entered, the system will automatically log the user off the existing number and onto the new account number on the same terminal and port and baud rate. If the account/password combination is incorrect the system will type:

   XXX.XX HOURS
   INVALID PASSWORD — NO TRANSFER

   and the user will be returned to the old account number with the current time used re-set to zero. Whether or not the new HELLO command was accepted by the computer, the user does not have to re-dial the system, and the HELLO message is usually limited to the word READY. The current program is retained in the user's swap track. Transferring from one account to another in this manner affords the user a very fast and convenient

way to transfer a program between account numbers without punching it out on paper tape and then reading it back.

For example,

```
HEL-V050,

000.01 HOURS


READY


GET-TAKEALONG
HEL-V090,

000.01 HOURS


READY


SAV
BYE
```

Should the user wish to log off an account by logging onto a different account, but not save the work being done on the first account, then the SCRatch command should be executed either immediately prior to the second log-on, or immediately following it.

3. A third method of logging off, valid only for terminals operating over telephone lines, is to hang up the telephone. The system automatic disconnect will sense the loss of carrier and will log the user off the system. The current program is lost and the user does not receive a written record of the time used. A user can, in some cases log off a directly connected ("hard wired") terminal in this manner or by turning the LINE/OFF/LOCAL switch to OFF or to LOCAL. (Users may occasionally be logged off the system on an unplanned basis because of temporary loss of telephone carrier or of power at the computer or at the terminal, or by spurious signals in the telephone lines. To minimize the inconvenience of such interruptions, it is prudent to SAVe a working copy periodically of the program currently in the work area.)

4. A forced log-off may be program-initiated by chaining to "$$$$". This technique is sometimes useful with non-abortable programs or with long programs that frequently run unattended where an automatic log-off is desired at completion of the program.

1.3 COMPUTER OPERATING MODES

The computer operates in any of three modes: the program entry mode, program execution mode, or Execute Immediate Mode.

1.3.1 Program Entry Mode

In program entry mode the computer analyses input lines to determine whether they are BASIC program statements (first character numeric) or direct computer commands (first character alphabetic).

1.3.2 Program Execution Mode

The computer is in program execution mode when a program is being run. This includes the program which is awaiting input from the user.

1.3.3 Execute Immediate Mode

The computer is in Execute Immediate Mode when an executing program is suspended without having executed an END statement. Specifically, there are three ways to enter Execute Immediate Mode:

1. The executing program encountered a STOP statement
2. A programming error was encountered (any program error which sends the system into Execute Immediate is called a terminal error)
3. The BREAK key was pressed once during program execution

The user will recognize that the computer is in Execute Immediate Mode because the system will type

```
STOPPED AT n
XI?
```

where n is the line in the current program which immediately follows the line where execution was interrupted. Several options are available at this point. One, the user can type GOTO n where n is a line number in the current program, and resume execution. Two, the user may execute a BASIC program statement, or three, the user may terminate the currently suspended program by typing END or by typing control C (C^C) or by typing a carriage return.

1.4 TERMINAL OPERATING CHARACTERISTICS

The BTI system supports a wide variety of ASCII terminals, and although operating conventions may vary slightly from one make or type of terminal to another, the procedures described below apply to the most commonly used terminals.

Every line of information entered into the computer through the terminal keyboard must be terminated by a carriage return. (Strike the RETURN key). The computer signals readiness to receive the next line of input by outputting a linefeed after the user-initiated carriage return.

Erroneously typed characters may be deleted by typing one backspace (← on some terminals or upper shift 0 (_) on some terminals) for each incorrectly typed character. This must be done prior to striking the RETURN key in order to be effective. For example, ACB←←BD←C (carriage return) would be the same to the computer as typing ABC (carriage return).

An entire line may be deleted by striking the ESCape or ALT-MODE key. The system will respond with a backslash (\), a carriage return and a linefeed, and will then be ready to receive the (correctly typed) line again. The ESC or ALT-MODE is used in lieu of the RETURN key in this instance.

Chapter 2

FUNDAMENTALS OF BASIC

2.1 NUMERIC REPRESENTATION

2.1.1 Numbers

The BTI system stores numbers in binary floating point form, and handles numbers in the range 5.87747×10^{-39} to 1.70141×10^{38}.

2.1.2 Significant Digits and E Notation

Decimal numbers to be input or output are limited to six significant digits. Numbers which exceed this limitation are rounded and output in 6-digit decimal scientific (or "E") notation. For example, the decimal value .001231237 is rounded to .00123124 (numbers less than 1 but greater than -1 are rounded beginning with the left-most non-zero digit) and is output as 1.23124E-03. Similarly, the decimal value 1231237 is rounded to 123124 and is output as 1.23124E+06, and the value 123123700 is rounded to 123124 and is output as 1.23124E+08.

2.2 ARRAYS

Array is the generic term for a numeric set. A one dimensional array is called a vector, and a two dimensional array is called a matrix.

In the BTI system, numbers may be manipulated individually as "scalars" and as individual elements of an array, or in groups, as arrays.

Where two dimensions are specified for an array, the one which is specified first (on the left) refers to the number of rows in the array, and the second, (the one on the right) refers to the number of columns. For example, the statement

```
0013 DIM A(3,4)
```

means that the array A is a matrix with 3 rows and 4 columns, and the statement

```
0013 DIM B(17)
```

means that the array B is a vector with 17 elements.

Unless it is defined to be otherwise, an array variable is assumed by the system to be either dimension 10 or dimension 10,10. The system resolves the ambiguity based on how elements of the array are defined.

For example, the statement

```
0013 X(2,1)=6
```

implies to the system that the array X is a 10 by 10 matrix, provided that no DIM statement has been declared to the contrary, whereas the statement

```
0013 X(6)=489
```

implies to the system that the array X is a 10 element vector, again, provided that no DIM statement has been declared to the contrary.

2.2.1 Array Subscripting

To access a single element in an array, the variable name must be subscripted by either one or two numbers in parentheses following the variable name. (The number of subscripts, one or two, depends on how many dimensions the array has, i.e., one subscript is legal for vectors and two subscripts must be used for a matrix.) For example, the expression A(2,3) refers to the element of an array which is in the second row and the third column. Therefore, if

$$A = \begin{bmatrix} 12 & 11 & 10 \\ 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

then A(2,3) = 7

Arrays may be used in arithmetic operations only as variables, and not as constants. To output an entire numeric array the MAT PRINT statement is used. For further information see Section 2.8, Outputting Data.

2.3 EXPRESSIONS

The word "expression" is used in this manual in the mathematical sense to denote one or more variables, constants and/or operators which when taken together are reducable to a numeric value.

2.4 ARITHMETIC AND LOGICAL OPERATORS

In BASIC the system evaluates an expression by first replacing each variable in the expression with its value and then performing the specified operations on the values.

BASIC statements are written on one line for each statement and without subscripting of the form A_1 or superscripting of the form A^2, or numerator/denominator formats of the form $\frac{A}{B}$. Arithmetic operations are performed according to a specific hierarchy. Below is a table of the BASIC arithmetic operator hierarchical sequence.

SEQUENCE	BASIC	STANDARD FORM	MEANING
1	↑	exponentiation	A↑B means A^B
2	NOT	~ negation	(NOT X) = 0 IF X ≠ 0 = 1 IF X = 0
3	*	x multiplication	A*B means A x B
3	/	÷ division	A/B means A ÷ B
3	&	binary AND	(A&B)=0 IF A=0 OR IF B=0 =1 IF A=1 AND IF B=1
3	%	binary OR	(A%B)=0 IF A=0 AND IF B=0 =0 IF A=1 AND IF B=1 =1 IF A=0 AND IF B=1 =1 IF B=1 AND IF B=0
4	+	+ addition	A + B
4	–	– subtraction	A – B
5	MIN	minimum	(A MIN B) = A IF A < B = B IF A ⩾ B
5	MAX	maximum	(A MAX B) = A IF A > B = B IF A ⩽ B
6	#	≠ not equal	(A # B) = 0 IF A = B = 1 IF A ≠ B
6	<=	⩽ less than or equal	(A <= B) = 0 IF A > B = 1 IF A ⩽ B
6	>=	⩾ greater than or equal	(A >= B) = 0 IF A < B = 1 IF A ⩾ B
6	<	< less than	(A < B) = 0 IF A ⩾ B = 1 IF A < B
6	>	> greater than	(A > B) = 0 IF A ⩽ B = 1 IF A > B
7	AND	logical and	(A AND B) = 0 IF A=0 OR IF B=0 = 1 IF A≠0 AND B≠0
8	OR	logical or	(A OR B) = 0 IF A=0 AND B=0 = 1 IF A≠0 OR IF B≠0

Between any two operators in the same BASIC statement, the one higher in the above table is performed first. If they are at the same level (# and >= for example) they are performed from left to right as they occur in the statement.

An arithmetic operator is one whose result reflects some interaction between the operands. (Operators at the 1, 3 and 4 levels are arithmetic operators.) A logical operator is one whose result reflects some relationship between the operands. (Operators at the 2 level and at the 5 through 8 level are logical operators.)

2.5 VARIABLES

In BASIC-X, the allowed types of variables and variable names are,

VARIABLE	NAME	EXAMPLE
numeric simple (or "scalar" or one-element) variable	letter or letter+digit (digit 1 – 9)	N T6
numeric array (multi-element) variable	letter	A
character string (one- or multi-element) variable	letter+$	Z$

The same letter may be used to name any or all of the above types of variables in one program. For example,

```
LIS
ØØ1Ø     DIM A$[15],A[3,3]
ØØ2Ø     A$="STRING VARIABLE"
ØØ3Ø     MAT READ A
ØØ4Ø     DATA 1,2,3,4,5,6,7,8,9
ØØ5Ø     LET A=1ØØ
ØØ6Ø     LET A1=2ØØ
ØØ7Ø     LET A7=3ØØ
ØØ8Ø     MAT PRINT A;
ØØ9Ø     PRINT A$;
ØØ1Ø     PRINT A;
Ø11Ø     PRINT A1;
Ø12Ø     PRINT A7;
Ø13Ø     PRINT A[2,3];
9999     END

RUN

  1     2     3

  4     5     6

  7     8     9

STRING VARIABLE 1ØØ    2ØØ    3ØØ    6
DONE
```

Note that the letter A is used to name an array dimensioned 3,3 (in Line 10), to name a string variable A$ (in Line 30) and to name three simple variables, A, A1, and A7 (in Lines 50, 60 and 70 respectively). While in general it is recommended that different letter names be used in order to avoid confusion, the user should realize that he has at his disposal 260 simple variable names, 26 array variable names and 26 string variable names, any combination of which may be used in a single program.

Where a value or values have not been explicitly assigned, each type of variable is handled in a different way.

If no value has been assigned to the numeric variable A, then the statement,

```
0010  PRINT A
0020  END
RUN
```

will cause the computer to respond,

```
UNDEFINED VALUE ACCESSED IN LINE 10
```

and the system will go into Execute Immediate Mode.

```
STOPPED AT 20
XI?
```

Similarly, the statement

```
0010  MAT  PRINT A
0020  END
RUN
```

will cause the computer to respond,

```
ARRAY OF UNKNOWN DIMENSIONS
```

This error is caught during the compile phase of program execution, so the system stops compiling at the point where the error is encountered and returns to the program entry mode.

Just as the dimension(s) of an array must be declared, so must all of the elements in the array be defined. For example,

```
0010  DIM A[5]
0020  A[1]=A[2]=A[3]=1
0030  MAT  PRINT A
0040  END
RUN
```

now the system responds,

```
UNDEFINED VALUE ACCESSED IN LINE 30


STOPPED AT 40
XI?
```

because some values of the array A are, in fact defined but some as yet are not, the array cannot be dealt with as an entity.

If no value has been assigned to the string variable A$, then the statement

```
0010  PRINT A$
0020  END
RUN


DONE
```

will not cause an error. This is because unlike numeric variables, string variables are considered to be defined as being empty when they have not been assigned a string value. In the case cited above, a single carriage return would be output to denote that A$ is an empty string variable.

All variables are assigned values in one of three ways: the LET statement, the READ statement, or the INPUT statement.

2.5.1 LET and Implied LET Statement

SYNTAX

A. 0013 LET *variable=expression*

B. 0013 *variable=expression*

FEATURES

1. The LET statement assigns the value of an expression to a variable. For example, the statement

   ```
   0013 LET A=B=C=D=1
   ```

   is a legal way of assigning the value 1 to the variables A,B,C and D.

2. The descriptive verb (LET) may be omitted from this statement type and only from this statement type. The system recognizes any such verbless statement as an implied LET statement. For example,

   ```
   0010  DIM A[2,3]
   0020  A[1,2]=(B+C)/2
   0030  X=Y=Z[I,J]=-1
   ```

   LET statements are compiled more rapidly when the verb is used explicitly.

3. If an equal sign (=) is used in such a way that its meaning could be ambiguous then the system will assume that it is an assignment operator. For example, in statement 30 above, all three equal signs are treated as assignment operators. In the following case, however,

   ```
   0013  X=Y=(Z[I,J]=-1)
   ```

the value of X and Y depends on the value of the logical expression (Z(I,J)=-1). If the expression evaluates as "true" then X and Y will both have the value 1. If Z(I,J) does not equal -1, (i.e., if the expression evaluates as "false"), then X and Y will both have the value 0.

2.6 STRINGS

BASIC-X provides the user with the capability to manipulate character strings. String manipulation is accomplished by the use of string variables, string LET statements, and string IF statements. Strings may also be used in READ, INPUT and PRINT statements.

2.6.1 String Variables

FEATURES

1. String variable names must be of the form: letter$, e.g., Z$.
2. The specified dimension of a string (in a DIM statement) is said to be its physical length.
3. If it is not specified in a DIM statement, a string's physical length is taken to be 1.
4. The number of characters actually in the string is said to represent its logical length.
5. A string may contain fewer characters or the same number of characters as its physical length, but may never contain more characters than its physical length.
6. A string may be dimensioned from 1 to 254 inclusive, and may actually contain from 0 to 254 characters.

All string variables are considered to be arrays of characters, i.e., each array element is a single character. The logical length of the string is initialized to 0 (empty) at the beginning of program execution.

2.6.2 Subscripted String Variables

FEATURES

1. A string variable may have none, one or two subscripts. For example, assume that A$="ABCDEFG", then

PRINT A$	yields	ABCDEFG
PRINT A$(4)	yields	DEFG
PRINT A$(2:6)	yields	BCDEF

Where no subscript is specified the entire logical string is printed. Where one subscript is specified — A$(4) for example — the string beginning with the 4th character in this case,

and ending with the last character in the string is printed. Where two subscripts are specified — A$(2:6) for example — the string beginning with the 2nd character in this case and ending with the 6th character is printed.

2. Subscripts must have a value of at least one.

If two subscripts are specified, the second one in the pair must be no smaller than one less than the first, and if it is larger than the logical length of the string, the string is considered to be followed by an infinite number of blanks. For example, assume that A$="ABCDEFGH",

```
A$(5:10)    = "EFGH "
A$(11:10)   = "" (null string)
A$(0:1)     is illegal
A$(8:6)     is illegal
A$(9)       = "" (null string)
A$(10)      is illegal
A$(19:12)   is illegal
```

2.6.3 Destination Strings

A destination string is a string variable into which a different (source) string is being copied. Part or all of the destination string may be replaced by part or all of the source string. For example, if

```
0010 DIM A$(26),B$(10),C$(10)
0020 B$="ABCDE"
0030 C$="AKLMNOB"
```

then progressively,

1.	A$(1:5)=B$	yields	A$=ABCDE
2	A$(6:10)="FGHIJ"	yields	A$=ABCDEFGHIJ
3	A$(11:15)=C$(2:6)	yields	A$=ABCDEFGHIJKLMNO
4	C$=B$	yields	C$=ABCDE
5	B$=A$(1:10)	yields	B$=ABCDEFGHIJ
6	B$(3)=A$(10:12)	yields	B$=ABJKL

FEATURES

1. The destination string (to the left of the assignment operator (= sign)) must be large enough to hold the source string.

2. If no subscripts are specified (as in example 4 above) then the entire source string replaces the entire string in the destination variable.

3. If one subscript is specified for the destination string (as in example 6 above) then the destination string, beginning with the specified character is replaced by the specified source string.

4. If two subscripts are specified for the destination string (as in examples 1, 2 and 3 above) then they must satisfy all the rules given under the section on Subscripted String Variables. Additionally, the first subscript of the destination variable must be no more than one larger than the current logical length of the destination variable. The second subscript of the destination variable must be no greater than the physical length of that variable.

5. If the source string is longer than the destination then the source string is truncated on the right. If the source string is shorter than the destination, trailing blanks are appended as necessary.

2.6.4 LEN Function

gives the logical length of a string in characters. For example,

```
0010  DIM A$[254]
0020  A$="ABC"
0030  PRINT A$,
0040  PRINT LEN(A$)
0050  A$[4]="XYZ"
0060  PRINT A$,
0070  PRINT LEN(A$)
0080  END
RUN

ABC               3
ABCXYZ            6

DONE
```

2.6.5 String LET Statement

assigns the specified characters to a string variable. The verb LET is optional. For example,

String LET Statement	Value of A$
0010 LET A$="XYZ"	"XYZ"
0020 A$(LEN(A$)+1)=A$	"XYZXYZ"
0030 A$=A$(2)	"YZXYZ"
0040 LET A$(2:4)=A$(3:5)	"YXYZZ"

2.6.6 String IF Statement

compares two strings character by character, using their ASCII equivalents. String variables may be subscripted in an IF statement. The first difference between the two strings determines their relation and if one string ends before a difference is found, the shorter string is considered to be smaller. For example, if A$="ABC" and B$="ABCD" then in the following example statements 40 through 100 test various kinds of relationships between A$ and B$.

```
0010  DIM A$[10],B$[10],Z$[1]
0020  READ A$,B$
0030  DATA "ABC","ABCD"
0040  IF A$ <= B$ THEN  PRINT "LINE 40 TRUE"
0050  IF A$>B$ THEN  PRINT "LINE 50 TRUE"
0060  IF A$=B$ THEN  PRINT "LINE 60 TRUE"
0070  IF A$[10:10]=" " THEN  PRINT "LINE 70 TRUE"
0080  IF A$[10:10]=B$[10:10] THEN  PRINT "LINE 80 TRUE"
0090  Z$=""
0100  IF A$[10:10]=Z$ THEN  PRINT "LINE 100 TRUE"
0110  END
RUN


LINE 40 TRUE
LINE 70 TRUE
LINE 80 TRUE

DONE
```

Compound string IF statements of the form

```
0013 IF A$(1:1)="A" AND B$(1:1)="A" THEN 1000
```

are not allowed.

2.6.7 Strings in READ, DATA, PRINT and INPUT Statements

String variables may be freely mixed with numeric variables in READ, PRINT and INPUT statements.

When used in DATA statements or in response to INPUT statements, they may be mixed, but they must occur in the proper positions as defined in their corresponding READ or INPUT statements, and they must be enclosed in quotation marks.

There is one exception to this rule, when a string and only a string is to be entered on an input line. Even in this case if the string is to contain leading blanks then it must be enclosed in quotation marks. For example,

```
0010  DIM A$[20],B$[20],C$[20]
0020  READ C$[1:13]
0030  DATA "NO. MARY AVE."
0040  INPUT A$,B1
0050  INPUT B$
0060  PRINT A$;B$;B1;C$
0070  END
RUN


?"BASIC ",650
?TIMESHARING, INC.
BASIC TIMESHARING, INC. 650  NO. MARY AVE.

DONE
```

2.7 INPUTTING DATA

2.7.1 READ and DATA Statements

SYNTAX

0013 READ *variable,variable,variable*

0023 DATA *constant,constant,constant*

FEATURES

1. A READ statement is used to supply data to the program.
2. The READ statement contains a list of variable names which are to be assigned values. These values are obtained from the DATA-list.
3. The DATA-list consists of all the values contained in all the DATA statements in the program, linked together into a single list of data constants.

When a READ statement is executed new data is taken from the DATA-list. A pointer to this list is moved to point at the next constant when a value is assigned to a variable by a READ statement. The programmer must take care to order the constants in the DATA statements to correspond with the variables in the READ statements both in data type and in dimension. In the example below, the values 1 and 2 are assigned to the variables I and J respectively, the value of -2.5 is assigned to the array variable element A(1,2), and the value "ABCDEF" is assigned to the string variable Z$.

```
0010  DIM A[2,2],Z$[11]
0020  READ I,J,A[1,2],Z$
0030  DATA 1,2
0040  DATA -2.5
0050  DATA "ABCDEF"
0060  PRINT I;J;A[1,2];Z$[1:6]
0070  END
RUN

 1      2     -2.5          ABCDEF

DONE
```

2.7.2 RESTORE Statement

It is sometimes convenient to use the same data from a DATA Statement more than once. This may be done by resetting the pointer, via a RESTORE statement.

SYNTAX

A. 0013 RESTORE

B. 0013 RESTORE *linenumber*

FEATURES

1. Syntax A restores the pointer to the first data item in the first DATA statement in the program.

2. Syntax B restores the pointer to the first data item in the DATA statement specified by the line number cited in the RESTORE statement.

The line number specified in Syntax B does not necessarily have to contain a DATA statement. If it does not, then the pointer will be set to the first data item in the first DATA statement following the line number specified.

2.7.3 INPUT Statement

SYNTAX

A. 0013 INPUT *variable,variable*

B. 0013 INPUT *"string",variable,variable*

FEATURES

1. The INPUT statement seeks the values of the referenced variables from terminal keyboard input, as READ statements do from DATA statements.

2. Data types and dimensions may be freely mixed in INPUT statements and the keyboard input values must match the INPUT statement variables both in type and dimension.

When an INPUT statement is executed the system generates a prompt and then awaits the user's input. If Syntax A is used the prompt is a question mark. For example,

```
0010  DIM J$[254],K$[254]
0020  PRINT "TYPE TWO SAMPLE STRINGS. ";
0030  INPUT J$,K$
0040  PRINT "*";J$;"*";" *";K$;"*"
0050  END
RUN

TYPE TWO SAMPLE STRINGS. ?"FIRST STRING","SECOND STRING"
*FIRST STRING* *SECOND STRING*

DONE
```

Syntax B requires that each input string be entered on a separate line. This enables the user to include embedded quotation marks and leading blanks in an input string, since the string itself need not be enclosed in quotation marks. For example,

```
0010  DIM J$[254],K$[254]
0020  INPUT "TYPE TWO SAMPLE STRINGS. ",J$,K$
0030  PRINT "*";J$;"*";" *";K$;"*"
0040  END
RUN

TYPE TWO SAMPLE STRINGS. A "QUOTED" WORD
??"A QUOTED STRING"
*A "QUOTED" WORD* *"A QUOTED STRING"*

DONE
```

2.8 OUTPUTTING DATA

BASIC-X provides the user with a wide range of output formatting capabilities. This section discusses these capabilities, beginning with the easiest to use and progressing to those which offer the user maximum control over output formats.

The major output formatting features covered in this section are:

1. The PRINT statement
2. The MAT PRINT statement
3. The CARriage command
4. The TAB function
5. The CHR$ function
6. The PRINT USING statement
7. The IMAGE statement

2.8.1 PRINT statement

SYNTAX

0013 PRINT *expression,expression*

FEATURES

1. A PRINT statement causes the data following the word PRINT to be output at the terminal and formatted according to certain rules.
2. Expression is a BASIC expression.
3. Data types may be mixed within a PRINT statement expression list.
4. String constants must be enclosed in quotation marks.
5. Each expression must be followed by a comma or a semicolon separator, with two exceptions:
   A. string constants need no separators since they are already separated by the quotation marks which enclose them
   B. the last expression in a data list may have no separator following it, and
6. A separator which follows the last expression has a special purpose in a PRINT statement.
7. Comma and semicolon separators may be mixed within a PRINT statement expression list.

When it refers to printing, the word "default" means that the output format has not been explicitly specified by the user. Any PRINT or MAT PRINT statement is a statement for which a default output format will be used, while any PRINT USING statement results in output whose format has been specifically designed by the user.

2.8.2 Field Widths

When the system encounters a PRINT statement, for each expression to be output, the carriage is spaced to the right to the next closest field's first print position. Output of the expression begins in that position and fills the next N character positions where N is the number of characters or digits in the item. If the carriage is already in the first print position of a field, then printing begins in that position. Output data items are left justified in their fields when the PRINT statement is used.

2.8.3 Comma Delimiters

When all data items are separated (delimited) by commas, the computer breaks the carriage into 15-character fields for output. Each data item is printed in a separate field. The table below shows the first and last character position and the width of the fields on a standard width teleprinter carriage (72 characters) and on a wide (132 characters) teleprinter carriage. Note that on both types of carriages the last field is only 12 characters wide.

	1	2	3	4	5	6	7	8	9
Narrow Carriage — 72 Characters									
First Position	0	15	30	45	60				
Last Position	14	29	44	59	71				
Characters	15	15	15	15	12				
Wide Carriage — 132 Characters									
First Position	0	15	30	45	60	75	90	105	120
Last Position	14	29	44	59	74	89	104	119	131
Characters	15	15	15	15	15	15	15	15	12

2.8.4 Semicolon Delimiters

When the semicolon delimiter is used the resulting output is packed. In the case of strings, the semicolon delimiter causes the strings to be output end to end. In the case of numeric expressions, the rules for packed output are more complex, although as for packed strings, the field widths for packed numbers depend on the sizes of the numbers.

A. As is true with comma delimiters, all numerics followed by semicolon delimiters are printed with the first character position reserved for a minus sign. If the number is positive, the first character position in the field is left blank.

B. Integers of one, two or three digits are packed in six-character fields.

C. Integers of four or five digits are packed in nine-character fields.

D. Integers of six digits are packed in twelve-character fields.

E. Non-integers are packed in twelve-character fields.

F. Numbers of greater than six significant digits are dealt with later in this chapter.

The results produced when printing strings separated by commas and by semicolons are contrasted below.

```
LIS
0010  DIM A$[10]
0020  A$="ABCDEFGHIJ"
0030  PRINT A$,A$,A$
0040  END
RUN

ABCDEFGHIJ      ABCDEFGHIJ      ABCDEFGHIJ

DONE
30PRINT A$;A$;A$
LIS

0010  DIM A$[10]
0020  A$="ABCDEFGHIJ"
0030  PRINT A$;A$;A$
0040  END
RUN

ABCDEFGHIJABCDEFGHIJABCDEFGHIJ

DONE
```

The first time the program is run, commas are used in LINE 30 to separate the three iterations of A$. Each occurrence of A$ is printed in a discrete 15-character field. LINE 30 is then modified to use semicolon separators and the program is listed, then run again. This time each occurrence of A$ is printed in its own 10-character field. Had there been three different strings of three different lengths to be printed in LINE 30, each string would have been printed in a field size that exactly matched its particular length.

Consider the case where the strings to be printed are too large for a default 15-character field:

```
0010  DIM A$[16]
0020  A$="ABCDEFGHIJKLMNOP"
0030  PRINT A$,A$,A$
0040  END
RUN

ABCDEFGHIJKLMNOP                ABCDEFGHIJKLMNOP          ABCDEFGHIJKL
MNOP

DONE
```

Where the comma delimiters are used, the 16th character of A$ spills into the adjacent 15-character field on the right, so the next occurrence of A$ must begin in the first print position of the next available 15-character field. Contrast this with the output result where semicolon delimiters are used:

```
30 PRINT A$;A$;A$
RUN

ABCDEFGHIJKLMNOPABCDEFGHIJKLMNOPABCDEFGHIJKLMNOP

DONE
```

2.8.5 Terminating Delimiters

The phrase "terminating delimiter" refers to the delimiter which follows the last expression in a PRINT statement. If a terminating delimiter is present, it suppresses output of the linefeed and carriage return. The system will output linefeed and carriage return when printing reaches the logical end of the current line. Logical line length is subject to user control. See the CAR-n command.

Since it is legal to mix data types within a single PRINT statement expression list, the following program yields,

```
0010  DIM B$[5],C$[5],F$[6]
0020  READ A,D,F,B$,C$,F$
0030  DATA 2,4,8,"FOUR ","PLUS ","TIMES "
0040  PRINT B$;C$;B$;"=";F
0050  PRINT A;F$;A;"IS";D;
0060  PRINT "AND";
0070  PRINT D;F$;D;"EQUALS ";D*D
0080  END
RUN

FOUR PLUS FOUR = 8
 2    TIMES  2    IS 4    AND 4    TIMES  4    EQUALS  16

DONE
```

2.8.6 MAT PRINT Statement

The MAT PRINT statement is used to output numeric arrays.

SYNTAX

0013 MAT PRINT *array*, *array*

FEATURES

1. The MAT PRINT statement causes the variable(s) following the word PRINT to be output at the terminal in a column and row format which represents the shape of the array.
2. Array must be a numeric array variable.
3. Arrays must be delimited by commas or by semicolons except that the last array in the list need not have a delimiter following it.
4. The last array in the list may be followed by a comma or a semicolon, as in the PRINT statement data list.
5. If no delimiter follows the last array in the list, MAT PRINT handles the item as though a comma followed.
6. Arrays are printed with one blank row between each array row.

In the example below, the comma delimiter is implied by the absence of a delimiter, and yields the 15-character field format, one field for each column, and one line for each row.

LIS

```
0010  DIM A[2,3],B[2,3]
0020  Z=1
0030   FOR I=1 TO 2
0040    FOR J=1 TO 3
0050    A[I,J]=Z
0055    B[I,J]=Z+10
0060    Z=Z+1
0070    NEXT J
0080   NEXT I
0090  MAT  PRINT A,B
0100  END
```

```
RUN

 1              2              3

 4              5              6

 11             12             13

 14             15             16

DONE
```

In the next example, the same output format is implied, but there are more elements in each row than can be printed on one line. The system outputs a carriage return and a single linefeed and continues to print the remaining elements in the current row. After the last element in the current row has been printed, the system outputs one carriage return and two linefeeds and begins to print the next array row.

```
0010  DIM A[3,6]
0020   FOR I=1 TO 3
0030    FOR J=1 TO 6
0040    A[I,J]=I+(J-1)
0050    NEXT J
0060   NEXT I
0070  MAT  PRINT A
0080  END

RUN

 1              2              3              4              5
 6

 2              3              4              5              6
 7

 3              4              5              6              7
 8

DONE
```

2.8.7 Logical Line Length

Logical line length refers to the number of characters to be output on one line by one or more PRINT statements. In the case where the length of an output line is greater than the logical carriage width, the system generates a carriage return delay as it does at the end of any line of output. For example,

```
0010  DIM A$[254]
0020  A$="THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG'S BACK"
0030  PRINT A$
0040  END

CAR-72
RUN


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

DONE

CAR-40
RUN


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

DONE
```

2.8.8 TAB Function

SYNTAX

0013 PRINT *expression*;TAB(*X*);*expression*

FEATURES

1. The TAB function moves the carriage to the right to the character position specified by the value of X.
2. X is an integer expression.
3. X must be less than the physical carriage width, or the logical carriage width, whichever is smaller.
4. The TAB function inserts the number of blank spaces necessary to move the carriage to the required position, and will not move the carriage backwards.

The TAB function permits forward placement of the carriage to a character position of the user's choice, subject to the rules above. X represents the absolute character position to which the carriage will space. The left-most character position is number 0.

```
LIS
0010  DIM Z[3,3]
0020  MAT  READ Z
0030  DATA 17,18,26,14,72,3,8,105,39
0040  MAT  READ Y[1,3]
0050  DATA 123456.,234567.,345678.
0060  PRINT "HEADING 1";TAB(15);"HEADING 2";TAB(30);"HEADING 3"
0070  PRINT "---------";TAB(15);"---------";TAB(30);"---------"
0080  PRINT
0090  MAT  PRINT Z,Y
0100  END
RUN

HEADING 1       HEADING 2       HEADING 3
---------       ---------       ---------

 17              18              26

 14              72              3

 8               105             39

 123456.         234567.         345678.

DONE
```

2.8.9 CHR$ Function and ASC Function

The CHR$ function converts its numeric argument to the corresponding character on the ASCII keyboard. The syntax is,

A. 0013 PRINT CHR$(X)

B. 0013 PRINT USING specifier;expression,CHR$(X)

FEATURES

1. The CHR$ function can only be used in PRINT and PRINT USING statements.
2. X is a numeric constant, variable or expression which is evaluated modulo 128 to the nearest integer.

The character which is output by CHR$ is determined by the value of the argument. For instance, the statement

0013 PRINT CHR$(74)

will output the letter J and the statement

0013 PRINT CHR$(42)

will output the * symbol. A table of the ASCII characters and their numeric representations appears in Appendix B.

The ASC function converts the first character of a string to its numeric ASCII code representation. The syntax is,

A. 0013 PRINT ASC(string)

B. 0013 X = ASC(string)

FEATURES

1. string may be a string constant enclosed in quotes, or a string variable, subscripted or not.
2. If string is of length zero, the ASC function will return the value –1.

2.8.10 PRINT USING and IMAGE Statements

PRINT USING and IMAGE are forms of the PRINT statement which give the user absolute control over output formats through a group of letters and symbols which are called format specifiers.

SYNTAX

A. 0013 PRINT USING "*specifier, specifier*";*expression,expression*

B. 0013 X$="*specifier,specifier*"
0023 PRINT USING X$;*expression,expression*

C. 0013 PRINT USING *XX*;*expression,expression*
00XX IMAGE *specifier,specifier*

FEATURES

1. PRINT USING outputs data according to a format specifier set which is declared by the user.
2. In SYNTAX A the format specifier string is enclosed in quotation marks in the PRINT USING statement.
3. In SYNTAX B the format string is stored in a string variable referenced by the PRINT USING statement, elsewhere in the program.
4. In SYNTAX C the format string is defined in an IMAGE statement referenced by the PRINT USING statement, elsewhere in the program.
5. A semicolon always separates the format string from the expression list.
6. Expression may be any numeric expression or a string variable, but may not be a string constant.

The format specifiers have implicit data type, and the data type must match that of the data item being formatted. In addition, for every data item to be formatted, there must be a format specifier set to accommodate the output.

SPECIFIER	RESULT
for strings	
X	Inserts a space between characters in a string.
A	Outputs one character of a string.
/	Outputs an X-OFF, Carriage Return, Linefeed.
for numerics	
.	Outputs a decimal point.
D	Outputs a numeric digit.
E	Converts a numeric value to E notation and provides four extra spaces for the exponent, e.g., E ±NN.
S	Outputs the sign of a numeric value.
X	Outputs a space between numeric digits.
Z	Outputs leading numeric zeros.
/	Outputs an X-OFF, Carriage Return, Linefeed.

for example,

```
0010  PRINT USING "DDDD.DD";399+702.02
0020  END
RUN

1101.02

DONE
```

In the example above, the specifier set "DDDD.DD" allows for up to four digits to the left of the decimal point and up to two digits to the right of the point.

Where there are too few digits provided for in the specifier string, the output value is converted to E notation.

```
LIS

0010  PRINT USING "DDD";4444
0020  END
RUN


+4.44400E+03

DONE
```

Where too few characters are provided for in formatting a string variable, the string is truncated.

```
ØØ1Ø     DIM J$[7]
ØØ2Ø     J$="EXAMPLE"
ØØ3Ø     PRINT USING 40;J$
ØØ4Ø     IMAGE AAAA
ØØ5Ø     END
RUN

EXAM

DONE
```

2.8.11 Replicators

A shorthand way of declaring multi-character or multi-digit specifiers is to use replicators. For example,

0013 PRINT USING "4D.3D";4607.235

means the same as 0013 PRINT USING "DDDD.DDD";4607.253.

2.8.12 Specifier Sets

A specifier set comprises all format specifiers necessary to output one formatted expression. Spaces to be inserted between data items by the X specifier may either be treated as separate sets, or may be appended to the beginning or the end of a set. Spaces to be inserted within an expression must be included within that specifier set.

Data items in a PRINT USING statement expression list may be selectively omitted from output.

For every data item which is to be omitted from the output, there must be a comma in the proper position in the specifier list. For example,

```
LIS


0010  READ A,B,C
0020  DATA 10,99,20
0030  DIM Z$[17]
0040  Z$="2D,X,      ,S2D.3D"
0050  PRINT "   DATA ITEM 'B' IGNORED - (";Z$;")"
0060  PRINT
0070  PRINT USING Z$;A,B,C
0080  Z$[6:10]="3DE,X"
0090  PRINT
0100  PRINT
0110  PRINT "   DATA ITEM 'B' INCLUDED - (";Z$;")"
0120  PRINT
0130  PRINT USING Z$;A,B,C
0140  END
```

```
RUN

   DATA ITEM 'B' IGNORED - (2D,X,      ,S2D.3D)
10 +20.000

   DATA ITEM 'B' INCLUDED - (2D,X,3DE,X,S2D.3D)
10 990E-01 +20.000
DONE
```

Line 40 of the program defines the specifier sets for the variables A and C, and in place of a specifier set for the variable B a comma appears. Line 50 prints a heading which shows the present form of the specifier string, Z$. Line 70 outputs the variables in the PRINT USING statement.

Line 80 includes in the specifier string a specifier set for the variable B (3DE,X), and Line 110 prints a second heading which shows the modified form of the specifier string, Z$. Line 130 is identical to Line 70. The only difference is that Z$ now contains a specifier set for the variable B.

Below are some examples of the format of output generated by PRINT USING format specifier sets.

```
LIS

ØØØ1     PRINT USING "DDD";123
ØØØ2     PRINT USING "ZDD";4
ØØØ3     PRINT USING "DDS";5
ØØØ4     PRINT USING "4D";-6
ØØØ5     PRINT USING "S3D";7
ØØØ6     PRINT USING "DSDD";-890
ØØØ7     PRINT USING "SDD.DD";3.25
ØØØ8     PRINT USING "DS4D";-37
ØØØ9     PRINT USING "DDX3D.DD";-1567.89
ØØ1Ø     PRINT USING "D.3DE";3.14159
ØØ11     PRINT USING "DE";378
ØØ12     PRINT USING "DDD";-444
ØØ13     PRINT USING "D/D";1,2
ØØ14     END
```

```
RUN

123
ØØ4
 5+
  -6
  +7
8-90
 +3.25
 -  37
-1 567.89
3.142E+ØØ
4E+Ø2

-4.44ØØØE+Ø2

1
2

DONE
```

Notice that the output from line 12 of the program is set off from other output by an extra line feed before and after it. This is because the format specifier set was insufficient by one character place to accommodate the data.

2.8.13 Numeric Formatting Rules

1. There must be at least one D specifier.
2. There must be only one S specifier (which may be omitted).
3. There must be only one . specifier (which may be omitted).
4. There must be only one E specifier (which may be omitted). If used, it must be the last character in the specifier set.
5. There may be any number of X specifiers (which may be omitted).
6. There may be any number of Z specifiers (which may be omitted).
7. There may be any number of / specifiers (which may be omitted).

2.8.14 Carriage Control

In addition to the ability precisely to format letters and numbers on the printed line, the user can control the format of lines on the printed page. The following are available:

SYMBOL	FUNCTION
+	suppresses LINEFEED
–	suppresses X-OFF and CARRIAGE RETURN
#	suppresses X-OFF, CARRIAGE RETURN and LINEFEED
N^C	outputs a LINEFEED
O^C	outputs a CARRIAGE RETURN
X^C	outputs a NULL
Y^C	outputs a FORM FEED

The first three symbols in the list, (+, – and #) are called Carriage Control Specifiers. The rules governing their use are:

1. If no Carriage Control Specifier is used, then PRINT USING statements output X-OFF, CARRIAGE RETURN, LINEFEED in that order, after each formatted line of output.
2. Use of a Carriage Control Specifier is not mandatory.
3. Only one Carriage Control Specifier is permitted in a format string.
4. If used, the Carriage Control Specifier must be the first character in the format string.

For example,

```
ØØ1Ø    E$=","
ØØ2Ø    DIM A$[28],B$[28],C$[7],Y$[4],Z$[3]
ØØ3Ø    A$="THIS SENTENCE IS AN EXAMPLE "
ØØ4Ø    B$="OF THE '#' CARRIAGE CONTROL "
ØØ5Ø    C$="SYMBOL."
ØØ6Ø    Y$="#28A"
ØØ7Ø    Z$="1ØA"
ØØ8Ø    READ E,F,G,H
ØØ9Ø    DATA 1ØØØ,2ØØØØ,3ØØØØØ.,4.E+Ø6
Ø1ØØ    PRINT USING Y$;A$
Ø11Ø    PRINT USING Y$;B$
Ø12Ø    PRINT USING Z$;C$
Ø13Ø    PRINT
Ø14Ø    PRINT "HERE IS AN EXAMPLE OF THE '+' SYMBOL:"
Ø15Ø    PRINT USING 16Ø;E,F,G,H
Ø16Ø    IMAGE+DX3D,3X,2DX3D,3X,3DX3D,3X,DX3DX3D
Ø17Ø    PRINT USING "A";TAB(1),E$,TAB(1Ø),E$,TAB(2Ø),E$,TAB(28)
        ,E$,TAB(32),E$
Ø18Ø    END
```

```
RUN

THIS SENTENCE IS AN EXAMPLE OF THE '#' CARRIAGE CONTROL SYMBOL.

HERE IS AN EXAMPLE OF THE '+' SYMBOL:
1,ØØØ    2Ø,ØØØ     3ØØ,ØØØ     4,ØØØ,ØØØ

DONE
```

The format control characters, N^C, O^C, X^C and Y^C permit output of a LINEFEED, a CARRIAGE RETURN, a NULL or a FORM FEED respectively, to occur anywhere in the formatted output. The rules governing their use are,

1. The format control character is a string and must either be enclosed in quotes or must be assigned to a string variable.
2. Format control characters used in the data item list of a PRINT USING statement require an "A" format specifier for each format control character used. For example,

```
ØØ1Ø     DIM Z$[2]
ØØ2Ø     DIM A$[5],B$[3],C$[2],D$[4]
ØØ3Ø     A$="THIS "
ØØ4Ø     B$="IS "
ØØ5Ø     C$="A "
ØØ6Ø     D$="TEST"
ØØ7Ø     REM ***** Z$="CONTROL N"
ØØ8Ø     Z$=" "
ØØ9Ø     PRINT USING 1ØØ;A$,Z$,B$,Z$,C$,Z$,D$
Ø1ØØ     IMAGE 5A,A,3A,A,2A,A,4A
Ø11Ø     END

RUN

THIS
    IS
      A
       TEST

DONE
```

3. Like all strings, the format control characters may be used in the data item list of a PRINT USING statement, only as string variables and not as literal strings, and they can be used in the format specifier list only in IMAGE statements. For example,

```
ØØ1Ø     DIM A$[4]
ØØ2Ø     A$="TEST"
ØØ25     REM ***** IMAGE 4A,"CONTROL N",4A,"CONTROL N, O",4A,"CONTROL
         N",4A
ØØ3Ø     IMAGE4A,"",4A,"",4A,"",4A
ØØ4Ø     PRINT USING 3Ø;TAB(5),A$,A$,TAB(3),A$,A$
ØØ5Ø     END

RUN

   TEST
       TEST
 TEST
     TEST

DONE
```

Since control characters are non-printing, Line 25 serves to inform the user which format control characters are used where in Line 30. The use of the TAB function in Line 40 is purely cosmetic.

2.9 STRING ARITHMETIC

The BASIC-X string arithmetic facility provides the user with the ability to store and manipulate decimal numbers. This is particularly useful when it is necessary or desirable to work with numbers having more than six significant digits. String arithmetic may be performed only in LET and implied LET statements.

SYNTAX

0013 LET *R*$=*M*$ *operator N*$

FEATURES

1. Strings may vary in length from 1 to 254 characters, inclusive.
2. Operand strings may contain any characters except that:
   a. only one sign (+ or –) is allowed per string operand
   b. only one decimal point (.) is allowed per string operand
   c. a string with no numeric characters is not a valid operand string
3. The numeric characters 0 through 9 are legal parts of string numbers.
4. A sign (+ or –) and/or a decimal point (.) may appear in any position.
5. Any characters other than those mentioned in items 3 and 4 above will be ignored and will not appear in the result.

When using string arithmetic the legal operators are = (assignment) + (addition), –(subtraction) and * (multiplication). (String division may be done by a utility program.)
For example,

```
Ø12Ø  A$=B$+C$
Ø13Ø  A$=A$*"2"
Ø14Ø  A$=C$-D$[1:5]
Ø15Ø  D$[7:1Ø]=D$[1:5]*"1.5"
Ø16Ø  D$=C$+P$[5]
Ø17Ø  M$=C$*D$[7]
```

The string variable in which the result of the operation will be stored must be dimensioned to be large enough to hold the entire result. If it is not large enough then the result string will be truncated on the right to fit the variable. This may change the magnitude of the result and no warning is given.

The first position of the result string is always either blank or a minus sign, and the result is always left justified with no leading zeros except that a decimal point is always preceded by a digit.

For addition and subtraction, the number of digits to the right of the decimal point in the result is equal to the greater number of digits to the right of the decimal point of the operands.

For multiplication, the number of digits to the right of the decimal point in the result is equal to the sum of the digits to the right of the decimal point in the two operands.

If there are no digits to the right of the decimal point then the decimal point is omitted. The result string ends immediately after the last numeric character.

2.9.1 VAL Function

SYNTAX

```
0013 R=VAL(Z$)
```

FEATURES

1. The VAL function converts numeric strings to numeric values.
2. R is a numeric scalar variable.
3. X$ is a numeric string.

Using the form of VAL shown above, the operand string may contain nothing but numeric characters and the optional "+", "–" and/or ".". Further, the operand string must conform to the rules for numeric strings.

It is possible to convert a string containing non-numeric characters, provided that the string conforms to the rules for numeric strings in all other respects. For example,

```
LIS


0010  DIM A$[25]
0020  A$="ABC+DE.FG123HIJ"
0030  Z=VAL(A$)
0040  PRINT Z
0050  END
RUN


BAD FORMAT FOR STRING NUMBER IN LINE 30


STOPPED AT 40
XI?END

DONE

23 PRINT A$
25 A$=A$+"0"
27 PRINT A$
LIS


0010  DIM A$[25]
0020  A$="ABC+DE.FG123HIJ"
0023  PRINT A$
0025  A$=A$+"0"
0027  PRINT A$
0030  Z=VAL(A$)
0040  PRINT Z
0050  END
RUN


ABC+DE.FG123HIJ
 0.123
 .123

DONE
```

The first version of the program tries and fails to convert the string A$="ABC+DE.FG123HIJ" to a numeric value, (LINE 30). In the second version, a string "0" is added to each character of A$ (LINE 25). This operation removes the non-numeric/non-arithmetic characters from A$ while performing the addition. The string result stored into A$ contains no extraneous characters.

2.10 MATRIX ARITHMETIC

NOTE: for a general discussion of arrays, see Section 2.2.

In order to handle all of the elements in an array as a single entity, the array must first be dimensioned and defined.

2.10.1 DIM Statement

The DIM statement declares the size and the dimensionality of an array.

SYNTAX

A. 0013 DIM X(*Z*)

B. 0013 DIM X(*Y*,*Z*)

FEATURES

1. The DIM statement reserves storage for arrays and sets upper bounds on subscripts. Arrays that do not appear in DIM statements are automatically given upper bounds of 10 or 10,10 the array being of one or two dimensions, respectively.

2. The bounds of an array may be changed dynamically in other matrix description statements such as MAT READ, so long as the storage requirement specified in these statements does not exceed that specified in the DIM statement for that array, or the 10,10 limitation, if no DIM statement is declared for the array.

3. Several arrays may be dimensioned by a single DIM statement, and several DIM statements may be used in a single program, but no single array may be cited in more than one DIM statement.

4. An array may be re-dimensioned as many times as required in MAT READ and MAT INPUT statements, within the constraints described further on in this chapter.

5. DIM statement parameters must be integer constants.

2.10.2 MAT READ and MAT INPUT Statements

Both the MAT READ and the MAT INPUT statements assign values to the elements of an array, and both may be used to declare the logical dimensions of an array.

SYNTAX — MAT READ

A. 0013 MAT READ X

B. 0013 MAT READ X(Y,Z)

FEATURES

1. Syntax A assumes that the array named X is dimensioned elsewhere in the program, and the MAT READ statement will seek in DATA statements the number of numeric values that the array is to contain. (This is stated in the most recent previous array declaration statement. For example, if the statement DIM X(4,5) were used, the MAT READ statement would look for 20 numeric elements to fill an array of 4 rows of 5 columns each.)

   Syntax B not only reads the values of X from DATA statements, it also assigns the logical dimensions of X. The logical dimensions of X may or may not have been declared previously in another statement.

   a. If the physical dimensions of X are declared in a DIM statement anywhere in the program, then the product of the dimensions of X in the MAT READ statement must not be greater than the product of the dimensions of X in the DIM statement, and the dimensionality of X must not change. That is, if X was declared in a DIM statement to be a one-dimensional array of 6 elements, then it cannot be declared in a MAT READ statement to be a two-dimensional array of any size. The reverse is also true. An example of this application of the MAT READ statement follows.

```
LIS


0010  DIM X[4,4]
0020  MAT  READ X
0030  DATA 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
0040  MAT  PRINT X;
0050  END
```

   b. Where the logical dimensions of an array are declared only in MAT READ or MAT INPUT statements, the array may be re-dimensioned as many times as necessary, to any dimensions within the 10 or the 10,10 constraint, but may not have their dimensionality changed. An example of this application of the MAT READ statement follows.

```
LIS


0010  DIM X[2,9]
0020  MAT  READ X[4,4]
0030  DATA 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
0040  MAT  PRINT X;
0050  END
```

3. Two dimensional arrays are filled by rows, that is, the top row is filled from left to right, then the next row down is filled from left to right, etc.

SYNTAX — MAT INPUT

A. 0013 MAT INPUT X

B. 0013 MAT INPUT X(Y,Z)

FEATURES

1. The MAT INPUT statement works exactly the same way as the MAT READ statement except that the values of the array are sought from terminal input instead of from DATA statements.

2.10.3 ZER Function

SYNTAX

A. 0013 MAT X = ZER

B. 0013 MAT X = ZER(Y,Z)

FEATURES

1. The ZER function works exactly the same way as the MAT READ statement except that it assigns numeric zeros to every element in the array without consulting a user-specified data list.

2. For example,

```
LIS


0010  MAT X=ZER[2,3]
0020  MAT  PRINT X;
0030  END
RUN

 0      0      0

 0      0      0


DONE
```

2.10.4 CON Function

SYNTAX

A 0013 MAT X = CON

B. 0013 MAT X = CON(Y,Z)

FEATURES

1. The CON function works the same way as the ZER function except that the array is filled with ones instead of with zeros.

2. For example,

```
0010  MAT X=CON[2,3]
0020  MAT  PRINT X;
0030  END
RUN

 1      1      1

 1      1      1

DONE
```

2.10.5 IDN Function

SYNTAX

A. 0013 MAT X = IDN

B. 0013 MAT X = IDN(Y,Z)

FEATURES

1. The IDN function works the same way as the ZER and CON functions with two exceptions:
   a. The matrix must be two dimensional and square (must have the same number of columns as it has rows)
   b. The IDN function assigns the identity matrix to the array variable

2. For example,

```
0010  MAT X=IDN[3,3]
0020  MAT  PRINT X;
0030  END
RUN
 1      0      0

 0      1      0

 0      0      1

DONE
```

2.10.6 Array Manipulation Statements

Once arrays have been dimensioned and their elements defined, they may be manipulated as single entities. BASIC-X provides the following programming statements expressly for handling arrays.

MAT X = M

copies one matrix into another matrix of the same dimensions. For example,

```
0010  DIM X[3,3],Y[3,3]
0020  MAT  INPUT Y
0030  MAT X=Y
0040  END
```

MAT X = L±R

does matrix addition or subtraction, element by element on matrices of the same dimensions.

For example,

```
0010  DIM X[3,3],L[3,3],R[3,3]
0020  MAT  INPUT L,R
0030  MAT X=L-R
0040  END
```

In both of the above statement types, the same matrix may appear on both sides of the equal sign.

MAT X = L*R

does matrix multiplication of L and R. For example,

```
0010  DIM X[2,5],L[2,4],R[4,5]
0020  MAT  INPUT L,R
0030  MAT X=L*R
0040  END
```

In this case the same matrix must not appear on both sides of the equal sign.

MAT X = INV(Y)

assigns the inverse of a square matrix to a different matrix of the same dimensions. For example,

```
0010  DIM X[3,3],Y[3,3]
0020  MAT  INPUT Y
0030  MAT X=INV(Y)
0040  END
```

MAT X = TRN(Y)

assigns the transpose of a matrix to a different matrix of appropriate dimensions. For example,

```
0010  DIM X[2,4],Y[4,2]
0020  MAT  INPUT Y
0030  MAT X=TRN(Y)
0040  END
```

MAT X = (*expression*)*Y

multiplies every element in the "Y" matrix by the result of the "expression" parameter and assigns it to a matrix of the same dimensions. In this case the same matrix may appear on both sides of the equal sign. For example,

```
0010  DIM X[2,3]
0020  A=B=14
0030  MAT  INPUT X
0040  MAT X=(A*(7/B))*X
0050  END
```

Chapter 3

PROGRAM STRUCTURE

The normal execution sequence of statements in a program is from the lowest to highest numbered statement. For a variety of reasons the user may want to alter this sequence. BASIC-X offers the programmer several convenient methods of modifying execution sequence.

3.1 END STATEMENT

No matter how the program execution sequence may have been modified, every program must have an END statement as its highest numbered line.

SYNTAX

9999 END

FEATURES

1. The END statement must appear at least once, as the highest numbered statement in any program.
2. The END statement may appear many times in one program to allow many logical exit points from the program.

Following are the methods at the programmer's disposal for altering the normal program execution sequence.

3.2 GOTO STATEMENT

A GOTO statement overrides the normal order of program statement execution by arbitrarily directing execution control to a line other than the next higher numbered statement in the current program.

SYNTAX

A. 0013 GOTO *linenumber*

B. 0013 GOTO *expression* OF *linenumber,linenumber*

FEATURES

1. Execution control is directed to the line number following the word GOTO in Syntax A, or one of the line numbers following the word OF in Syntax B.

2. It is legal to GOTO non-executable statements such as REM statements. If this is done then control passes to the first executable statement whose line number is greater than that specified by the GOTO statement.

3. The referenced line number must exist in the program and it is not legal to specify a line number greater than the highest numbered statement in the current program.

The multi-branch (or "computed") GOTO statement (Syntax B) permits a selection of statements to which execution control may pass, based on the evaluated result of the expression. If the expression evaluates to a non-integer, then the result is rounded.

The expression to the right of the word GOTO must have a result in the range 1 to N where N is the length of the linenumber list. If the result of the expression is greater than N or less than .5, the execution control passes to the statement immediately following the GOTO statement. For example,

```
0010  INPUT "B AND X?",B,X
0020  GOTO B+X^2 OF 50,70,90
0030  PRINT "NO BRANCH"
0040  GOTO 100
0050  PRINT "WENT TO 50"
0060  GOTO 100
0070  PRINT "WENT TO 70"
0080  GOTO 100
0090  PRINT "WENT TO 90"
0100  PRINT
0110  INPUT "MORE? ",A$
0120  PRINT
0130  PRINT
0140  IF A$="Y" THEN 10
0150  END
RUN


B AND X?0,0
NO BRANCH

MORE? Y


B AND X?0,1
WENT TO 50

MORE? Y
```

```
B AND X?1,1
WENT TO 70

MORE? Y


B AND X?2,1
WENT TO 90

MORE? Y


B AND X?2,2
NO BRANCH

MORE? N
```

Line 20 of the program contains the computed GOTO statement. There are three line numbers in the list following the word OF.

Line 30 is executed only if the expression B+X↑2 evaluates to less than .5 or greater than 3.

In five iterations of the program, the first and fifth iterations result in no branch. The first does so because 0+0↑2=0, and the fifth because 3+1↑2=4. Both of these results are outside the .5 to 3 range required by the three line numbers referenced in Line 20.

3.3 IF . . . THEN. . . STATEMENT

The IF. . .THEN. . . statement can best be described as a conditional GOTO statement. Like the GOTO statement, the IF. . .THEN. . . statement is used to modify the normal program statement execution sequence, but the modification is done on a conditional basis, that is, only if the condition specified by the expression actually exists.

SYNTAX

A. 0013 IF *expression* THEN *linenumber*

B. 0013 IF *expression* THEN *statement*

FEATURES

1. Expression may be any valid BASIC-X arithmetic or logical expression.

2. If the expression evaluates to zero then it is considered to be false, and if it evaluates to non-zero (positive or negative) it is considered to be true.

If Syntax A is used, a "true" expression causes execution control to pass to the specified line number. If "false", then control passes to the statement immediately following the IF. . .THEN. . . statement.

If Syntax B is used, a "true" expression causes the statement following the word THEN to be executed, and if "false", the statement following the word THEN is ignored, and control passes to the statement immediately following the IF. . .THEN. . . statement. For example,

```
0010  K=1
0020  INPUT "ENTER N ",N
0030  IF N#5 THEN 50
0040  K=10
0050  PRINT "N= ";N;"K= ";K
0060  END
RUN


ENTER N 3
N=  3     K=  1

DONE
RUN


ENTER N 5
N=  5     K=  10

DONE
```

3.4 FOR. . .NEXT. . . STATEMENTS

FOR and NEXT statements are used in conjunction to simplify the writing of program loops.

SYNTAX

0013 FOR V = B TO E STEP S

0023 NEXT V

FEATURES

1. The FOR statement specifies the counting variable (V above), its numeric range (B TO E above) and the counting step size (STEP S above).

2. The NEXT statement increments the counting variable by the step size and then returns execution control to the matching FOR statement.

3. FOR. . .NEXT. . . loops may be nested to any desired level. For example,

```
0010  DIM M[5,4]
0020   FOR J=1 TO 5
0030    FOR K=4 TO 1 STEP -1
0040    M[J,K]=(J+K)
0050    NEXT K
0060   NEXT J
0070  MAT  PRINT M
0080  END
```

4. Every FOR statement must have a corresponding NEXT statement, and every NEXT statement must have a corresponding FOR statement.

The FOR. . .NEXT. . . loop employs the following algorithm:

```
0100 FOR V = B TO E STEP S
.
.
.
.
0200 NEXT V
```

1. Evaluate the beginning value and the ending value (B and E above, respectively) for the counting variable (V above).
2. Evaluate the step size if specified (S above) and if unspecified, then STEP = 1.
3. LET V (counting variable) = B (beginning value).
4. IF S > 0 AND V > E or IF S < 0 AND V < E then GOTO step 7.
5. Execute all statements between the FOR statement and its matching NEXT statement.
6. NEXT statement: LET V = V+S and then GOTO step 4.
7. Execute the statement following the NEXT statement.

3.5 GOSUB. . .RETURN statements

GOSUB and RETURN statements are useful when a certain subsection of a program must be executed repeatedly during a single execution of the program as a whole.

SYNTAX

A. 0013 GOSUB *xxxy*
xxxz RETURN

B. 0013 GOSUB X OF *xxxa,xxxb,xxxc*
xxxz RETURN

FEATURES

1. *xxxy* is a line number in the current program other than the line in which the GOSUB appears.
2. The subroutine section of the program must end logically with a RETURN statement.
3. To avoid disrupting the logical flow of the program as a whole, the subroutine must not be exited with any but a RETURN statement.
4. GOSUB's may be nested 9 deep.

The system retains the line numbers immediately following those on which GOSUB statements appear as addresses in a last-in-first-out stack. On encountering a RETURN statement, the system passes execution control to the most recent address in the stack.

A given subroutine may contain several RETURN statements. This allows for several logical exit points from the subroutine.

Syntax B behaves the same as a multi-branch GOTO statement and follows the rules given above for GOSUB.

3.6 ON ERROR . . . RESUME statements

The ON ERROR statement provides program controllable error handling capability. The syntax is,

0013 ON ERROR THEN *linenumber*

FEATURES

1. The ON ERROR statement sets a flag which causes execution control to pass to linenumber if an error is encountered. (A list of possible errors and their codes appears at the end of this Section.)
2. *linenumber* is assumed to be the first line of an error handling subroutine.
3. *linenumber* must be an integer constant.

The error handling subroutine is entered via the ON ERROR statement and is exited by the RESUME statement. The syntax is,

A. 0013 RESUME

B. 0013 RESUME *linenumber*

FEATURES

1. Syntax A returns execution control to the line in which the error occurred.
2. Syntax B passes execution control to *linenumber.*
3. *linenumber* must be an integer constant
4. Execution of a RESUME statement re-sets REF(6) and REF(7) to zero. (See description of REF function arguments 6 and 7 below.)
5. If a RESUME statement is executed before an ON ERROR statement, or after the ON ERROR flag has been revoked, the diagnostic message

   RESUME WITHOUT PRIOR ERROR

   will be generated.

The ON ERROR flag is revoked as follows:

0013 ON ERROR THEN 0

FEATURES

1. If the ON ERROR flag is revoked during execution of an error handling subroutine in an abortable program, the system will print the appropriate diagnostic message for the error encountered and then go to Execute Immediate Mode as though the error flag had never been set.
2. If the ON ERROR flag is revoked during execution of an error handling subroutine in a non-abortable program, the system will print the appropriate diagnostic message for the error encountered and then terminate the program by executing an END statement.

ON ERROR flags cannot be nested. If a second ON ERROR statement (other than ON ERROR THEN Ø) appears in the program, including in an error handling subroutine, the error flag will be reset for errors encountered subsequently, outside the error handling subroutine.

Pressing the BREAK key qualifies as an error. The system finishes processing the program statement it was working on when the BREAK key was pressed and then the ON ERROR flag takes effect.

If a BREAK occurs during execution of an error handling subroutine in an abortable program, the system will go to Execute Immediate Mode. If the BREAK occurs in a non-abortable program error handling subroutine, the system will ignore the BREAK and finish executing the subroutine. WARNING ONLY errors are not handled by the ON ERROR statement.

The REF function arguments 6 and 7 are used in conjunction with ON ERROR. REF(6) yields an error code and REF(7) yields the number of the line in which the error occurred. An example of how the REF function might be used follows:

```
    .
    .
    .
Ø113 ON ERROR THEN 7ØØ
    .
    .
    .
Ø213 READ #1,X:B$                    (possible End-Of-File/End-Of-Record error)
    .
    .
    .
Ø313 PRINT #1;Z,B$                   (possible End-Of-File/End-Of-Record error)
    .
    .
    .
Ø7ØØ IF REF(6)=5Ø THEN 8ØØ
Ø71Ø ON ERROR THEN Ø                 (revoke ON ERROR flag if error code ≠ 5Ø)
Ø8ØØ IF REF(7)=213 THEN 9ØØ
Ø81Ø PRINT "NO MORE ROOM IN FILE"
Ø82Ø RESUME 323
Ø9ØØ PRINT "NO MORE DATA IN FILE"
Ø91Ø RESUME 223
    .
    .
    .
9999 END
```

ERROR CODES RETURNED BY REF(6)

1	BREAK
2	timeout (4000-plus models)
3-9	reserved
10	undefined statement reference
11	NEXT without matching FOR
12	same FOR-variable nested
13	function defined twice
14	variable dimensioned twice
15	last statement not END
16	unmatched FOR
17	undefined function
18	array too large
19	array of unknown dimensions
20	out of storage
21	dimensions not compatible
22	characters after command end
23	invalid file mode
24	no such file
25	GOSUBs nested ten deep
26	RETURN without prior GOSUB
27	subscript out of bounds
28	negative string length
29	non-contiguous string created
30	string overflow
31	out of data
32	data of wrong type
33	undefined value accessed
34	matrix not square
35	redimensioned array too large
36	nearly singular matrix
37	LOG of negative argument
38	argument out of bounds
39	zero to zero power
40	negative number to real power
41	argument of SIN or TAN too big
42	bad file number
43	last input ignored, retype it
44	statement not IMAGE
45	non-existent file referenced
46	busy/protected file
47	no such program
48	chained program too large
49	write tried on read-only file
50	end-of-file/end-of-record
51	file abnormal/unavailable
52	I/O protocol error (special ports)
53	bad file name
54	bad file length
55	disk full
56	exceeds allocated space
57	directory full
58	duplicate entry
59	not a BASIC file
60	non-close file
61	file in use
62	RESUME without prior ERROR
63	common file declared twice
64	reserved
65	missing format specification
66	illegal or missing delimiter
67	no closing quote
68	bad character after replicator
69	replicator too large
70	replicator zero
71	multiple decimal points
72	bad floating point specification
73	illegal character in format
74	illegal format for string
75	missing right parenthesis
76	missing replicator

77 too many parenthesis levels

78 missing left parenthesis

79 illegal format for number

80 reserved

81 bad format for string number

3.7 LOCAL AND INTRINSIC FUNCTIONS

By definition, a function describes and solves for the mathematical relationship between two variables (such as X and Y), such that for each value of X there is one and only one value of Y. In other words, a function is a shorthand way of iterating a mathematical expression. For example,

```
0013 B=5
0014 X=(1+(B*2))/3
```

may be represented

```
0013 DEF FNZ(B)=(1+(B*2))/3
0014 Y=5
0015 X=FNZ(Y)
```

For each iteration of FNZ only the value of the argument (B) need be defined. For example,

```
0013  A=FNZ(2*1.8)
```

is a legal way to "call" the function FNZ.

In general the format for using a function in a program is,

```
0013 variable=function(argument)
        or
0013 PRINT function(argument)
```

The argument to a function must be a numeric expression with a single number as its evaluated result.

In BASIC-X there are two types of functions. They are called Local Functions and Intrinsic Functions.

3.7.1 Local Functions

Local or user-defined functions are defined in and are local to user-defined programs or parts of programs. The DEF statement is used to define a local function.

SYNTAX

0013 DEF FNX(Y) = *expression*

FEATURES

1. A local function may have only one parameter.

2. A local function is identified by a three-letter name, the first two letters of which must be FN. There may be up to 26 local functions per program, one for each letter of the alphabet, e.g., FNA, FNB,. . .FNY, FNZ.

3. For FNX(Y), Y is a dummy argument. When the function is actually used, the argument may be any expression.

4. Any operand which is available to the current program including references to existing functions may appear in the defining expression for a local function.

When a function is defined, e.g., 0013 DEF FNA(Z), the variable name in parentheses following the function name ((Z) in this case) is called the formal parameter, or sometimes the dummy parameter.

A function is "called" when it is referenced in a program, having already been defined in the program, e.g., 0013 A=FNA(Y+2).

When a function is called, e.g., 0013 A=FNA(Y+2), the expression in parentheses following the function name ((Y+2) in this case) is called the argument or sometimes the "actual parameter".

3.7.2 Intrinsic Functions

Intrinsic functions are parts of the language itself and are available to every user. The intrinsic functions available are:

FUNCTION	PURPOSE
ABS(X)	gives the absolute value of X
EXP(X)	gives e (2.718281) to the Xth power
LOG(X)	gives the natural logarithm of X
SQR(X)	gives the square root of X
SIN(X)	gives the sine of X
COS(X)	gives the cosine of X
TAN(X)	gives the tangent of X
ATN(X)	gives the arctangent of X
INT(X)	gives the largest integer $\leqslant$ X

FUNCTION	PURPOSE
SGN(X)	gives 1 if X > 0 0 if X = 0 –1 if X < 0
TYP(X)	gives the type of the next data to be read, either from a file or from a DATA statement. The use of the TYP function for file data is described in Section 6.8.3. For data to be read from a DATA statement data list, the syntax for the TYP function is 0013 A = TYP(0) There are three possible results: 1 = numeric data, 2 = string data and 3 = out-of-data condition.
RND(X)	gives a random number
RND(0)	selects a number from a source which is unique to the user's port
RND(–N)	selects a number from a source which is referenced by all system ports
RND(N)	selects from the unique port source, a specific number which is implied by the specific value of N. (There is no consistent relationship between the value of N and the value of the random number that N selects, other than that the same value of N will always cause the same number to be selected from the random number list.)

The selection of a random number is made from a list of about 8 million numbers >0 and <1. The list has 33 pointers: one common pointer for the system plus 32 different pointers, one for each of the 32 system ports.

When the system sees a 0 as the argument to RND, it references that port's pointer and makes a random selection from the list, beginning where the port pointer currently rests. For example,

```
0010  INPUT "HOW MANY SELECTIONS? ",Z
0020   FOR J=1 TO Z
0030   PRINT RND(0);
0040   NEXT J
0050  END
RUN

HOW MANY SELECTIONS? 5
 .648443      .979838      .510626      .223119      2.03642E-02
DONE
RUN

HOW MANY SELECTIONS? 9
 .464033      2.94644E-02     .289146      .288182      6.71268E-02
 .888236      .124827      .742629      .963338
DONE
```

Port pointers may be set to a specific position by declaring a specific positive argument to RND. Thus, if the parameter to RND is 6, say, then regardless of how many selections are made from the list, the selection will always be the specific number in the list which is associated with that specific value of the RND argument. For example,

```
0010  INPUT "HOW MANY SELECTIONS?",Z
0020   FOR J=1 TO Z
0030   PRINT RND(6);
0040   NEXT J
0050  END
RUN


HOW MANY SELECTIONS?4
 .250422      .250422      .250422      .250422
DONE
```

The following program permits a variable number of random number selections, but insures that every sequence of selections will be identical to all others, so far as it goes. (The value of .02 used in line 10 is completely arbitrary.)

```
0010  A=RND(.02)
0020  INPUT "HOW MANY SELECTIONS? ",Z
0030   FOR J=1 TO Z
0040   PRINT RND(0);
0050   NEXT J
0060  END
RUN


HOW MANY SELECTIONS? 4
 .980846      3.35703E-02     .16256       .503207
DONE
RUN


HOW MANY SELECTIONS? 5
 .980846      3.35703E-02     .16256       .503207      .168633
DONE
RUN


HOW MANY SELECTIONS? 6
 .980846      3.35703E-02     .16256       .503207      .168633      .349665
DONE
```

To output integers in the range 0 to 99, line 40 of the above program might be written,

```
0040 PRINT INT(RND(0)*100)
```

When the system sees a negative parameter to RND, it selects the number in the list where the system pointer currently happens to rest. Groups of random numbers selected in this way cannot be intentionally iterated. For example,

```
0010  A=RND(.02)
0020  INPUT "HOW MANY SELECTIONS? ",Z
0030   FOR J=1 TO Z
0040   PRINT RND(-1);
0050   NEXT J
0060  END
RUN


HOW MANY SELECTIONS? 4
 .306689      .537845      .696617      .940248
DONE
RUN


HOW MANY SELECTIONS? 5
 8.94691E-02    9.68938E-02    .64532       .17754       .288125
DONE
RUN


HOW MANY SELECTIONS? 6
 .105621      .118097      .150467      .295455      9.95302E-02     .889207
DONE
```

FUNCTION	PURPOSE
REF(X)	gives the current program access to various kinds of system information. There are seven parameter values to REF:
	1 — gives the current time of day in tenths of a second (0 to 863999).
	2 — gives the Julian date in the current year (1 to 366).
	3 — gives the last two digits of the current year (e.g., 76)
	4 — gives the user's currently logged on account number where the account letter is given numerically as the most significant digits, (i.e., A450 = 1450, D123 = 4123 or Z999 = 26999).
	5 — gives the port from which this REF is being executed (0 to 31).
	6 — gives error code (see ON ERROR statement).
	7 — gives error line number (see ON ERROR statement).
CHR$(X)	prints the ASCII character which is represented by the number X. X must be in the range 0—127. If it is outside this range, then it is converted modulo 128. For further information, see Section 2.8.9.
LEN(A$)	gives the current logical length of a string in number of characters. A$ must be a string variable.

FUNCTION	PURPOSE
TAB(X)	tabs to the character position specified by X. For further information, see Section 2.8.8.
VAL(A$)	gives the numeric value of digits in a character string. For further information, see Section 2.9.1.

Chapter 4

PROGRAMS

4.1 COMPILER OPERATION

Phase One — is active during program entry. As each statement is typed, the compiler checks the statement format and syntax and if correct, forwards the statement for inclusion in the current program. If the statement is incorrectly formatted then it is not included in the current program and a message is typed,

ERROR (no carriage return)

The user may type any single character such as a colon followed by a carriage return, and the system will respond with a more detailed diagnostic. If the error is obvious to the user, then the diagnostic may be aborted by striking only the RETURN key. The system is then ready for more input.

Phase Two — is active at run time (on the user's command that the program be executed). During phase two the system

1. allocates space for arrays
2. checks to see that statements such as GOTO and GOSUB, which modify the normal execution sequence, refer to valid line numbers in the program
3. checks to see that FOR . . . NEXT . . . statements form proper loops
4. checks for END as the highest numbered statement in the program

For errors found during phase two, diagnostics are issued, program execution is terminated and program entry mode control of the system is returned to the user. If no errors are found in phase two then control passes to phase three.

Phase Three — executes the program statements one by one beginning with the line number specified in the RUN command. If no line number was specified in the RUN command, then execution begins with the lowest numbered statement in the program. File statements are executed during phase three. This establishes file buffers and linkages. Errors found during phase three are referred to as terminal errors. These errors produce diagnostics and then place the system in Execute Immediate Mode where execution of the current program is suspended without executing an END statement. More information on how to use Execute Immediate Mode is given in Section 4.7.

4.2 PROGRAM STATEMENTS

A BASIC program consists of a group of BASIC statements. Each statement begins with an execution sequence number (or more simply stated, a line number), and each statement is typed on a separate line.

4.3 LINE NUMBERS

Line numbers are positive integers in the range 1 to 9999, and denote the order in which the various program statements will be executed when the program is run. Each statement describes, through its particular combination of key verbs and mathematical operators, one or more actions to be taken by the computer.

Statements need not be defined in the same order as their intended execution sequence. The computer will arrange the statements in ascending numerical order based on their line numbers, regardless of the order in which the statements were entered. If a statement is typed with the same line number as a statement already in the program, the most recent version of that line/statement will replace the earlier entry. To delete a statement entirely, type only its line number and strike the RETURN key.

4.4 ENTERING A PROGRAM

It is good practice to number statements in increments of at least ten. By defining only every tenth available line as a BASIC statement, convenient "spaces" in the program are available for insertion of new or previously omitted statements. For example,

```
0010 PRINT A$
0020 END
RUN


DONE
```

as the program is defined above, the computer will print a NULL when line 10 is executed. A statement (number 5 in this case) can be inserted to specify A$ to be the word EXAMPLE.

```
0005 A$="EXAMPLE"

LIS


0005  A$="EXAMPLE"
0010  PRINT A$
0020  END
RUN


STRING OVERFLOW  IN LINE 5


STOPPED AT 10
XI?END
```

This is still an error in the program, and causes the system to go to Execute Immediate. Execute Immediate is exited by typing a carriage return and then a line containing a DIM statement for A$ must be added to the program so that the value of A$ can be assigned.

```
0001  DIM A$[7]
0005  A$="EXAMPLE"
0010  PRINT A$
0020  END
RUN


EXAMPLE

DONE
```

Each of the four lines in the example above is a program statement. Note that the last statement (highest numbered statement) in the program is and must be an END statement.

4.5 RENumber COMMAND

At this point it is desirable to renumber the program statements in such a way that the chosen line numbering increment is again constant, without changing the execution sequence of the statements. This is done by using the RENumber command. For example,

```
0001  DIM A$[7]
0005  A$="EXAMPLE"
0010  PRINT A$
0020  END
```

```
REN-10,10,1,20
LIS

0010   DIM A$[7]
0020   A$="EXAMPLE"
0030   PRINT A$
0040   END
```

In general, the RENumber command is used to give program statements a constant numbering increment.

SYNTAX

REN-*new,increment,start,stop*

"New" will be the number of the first renumbered statement.

"Increment" will be the size of the numbering increment.

"Start" was the number of the first statement to be renumbered.

"Stop" was the number of the last statement to be renumbered.

All four parameters are optional and if omitted, their default values are,

REN-10,10,1,9999

If the parameters are specified in such a way as to threaten the existing sequence of the program statements, the system responds,

SEQUENCE NUMBER OVERLAP

Any line number references in the program such as those in GOTO statements, are changed to match the renumbered section of the program, if applicable.

4.6 LISTING AND RUNNING A PROGRAM

To review all of the program statements, the LISt command is used.

SYNTAX

LIS-*n*

Causes the statements in the current program to be listed in ascending numeric sequence. The n parameter is optional and, if omitted the program statements are listed starting with the lowest

numbered statement in the program. If the n parameter is specified, then the program is listed starting with the statement number n, or the next higher statement number if n does not exist in the current program.

To execute the program the RUN command is used.

SYNTAX

RUN-*n*

Causes the current program to be executed. The n parameter is optional and, if omitted, the program is executed beginning with its first statement. If the n parameter is specified, then the program is executed starting with the statement number n, or the next higher statement number if n does not exist in the current program.

Program execution may be suspended by striking the BREAK key once. The exception is if the program is awaiting user input. In this case program execution cannot be suspended. The system prints the remaining contents of the output buffer which could contain as many as 300 characters (30 seconds' worth on a 110 baud port) plus any line feed delays and/or carriage return delays that have been set. The system types

STOPPED AT *line*
XI?

XI? indicates that the system is in Execute Immediate Mode.

Program execution may be aborted (if the program has ABOrtable status) by striking the BREAK key twice. The exception is if a program is awaiting user input. In this case program execution cannot be aborted. The contents of the output buffer are lost, and the system types,

ABORT

and returns to program entry mode.

A program awaiting user input may be terminated by typing control C (C^C — hold down the CONTROL key and strike the letter C key) followed by carriage return.

4.7 DEBUGGING A PROGRAM

During program entry it is sometimes convenient for debugging purposes, to be able to test alternative programming methods. The Execute Immediate Mode gives the user the ability to try different statements without executing the program anew for each alternative to be tried. The user may activate Execute Immediate by including STOP statements at appropriate points in the program.

SYNTAX

A. 0013 STOP

B. 0013 IF *expression* THEN STOP

FEATURES

1. The STOP statement suspends program execution and enters Execute Immediate Mode.

Once the computer is in Execute Immediate, the user has three options:

1. Execution of the current (suspended) program may be resumed by typing

   GOTO linenumber

   where linenumber is any line in the current program. The computer would then be in program execution mode.

2. BASIC statements may be executed one at a time and without line numbers. The computer remains in Execute Immediate Mode until either option 1 or option 3 is implemented.

3. Execution of the remaining portion of the current program may be elided by typing

   END -carriage return-

   or

   C^{C} -carriage return-

   or

   carriage return

The computer would then be in program entry mode ready for entry of program statements, or program or command execution.

In Execute Immediate Mode the computer accepts any input as a BASIC statement without a line number, and such statements are executed immediately after they are entered. Only COM, DATA, DEF, DIM, FOR, REM and IMAGE statements are disallowed in Execute Immediate.

The usefulness of Execute Immediate is in its application as a program debugging tool. STOP statements may be placed at strategic points in a program and when execution is suspended the values of program variables may be checked, the contents of files and locations of file pointers verified, or alternative program statements tested. Program statements in Execute Immediate are not preceded by line numbers, and any variable names used in Execute Immediate must already have been mentioned in the currently suspended program. For example,

```
0010  A=B=C=D=E=F=1
0020  STOP
0030  END
RUN



STOPPED AT 30
XI?PRINT A+B
 2
```

```
XI?A=6

XI?PRINT A+B
 7

XI?Z=1
ERROR; UNDEFINED SYMBOL

XI?PRINT Z+A
ERROR: UNDEFINED SYMBOL

XI?END

DONE
```

All BASIC statements except those mentioned above are legal in Execute Immediate. A GOSUB statement in Execute Immediate will cause the referenced subroutine in the current program to be executed and on encountering the RETURN statement, will revert to Execute Immediate.

Some examples of how Execute Immediate is commonly used are,

```
0010  DIM A$[254],Z[12]
0020  INPUT A$[1:10],Z[2]
0030  PRINT Z[6]
0040  PRINT A$[12:15]
0050  END
RUN


?"HEADING LINE",783

UNDEFINED VALUE ACCESSED IN LINE 30


STOPPED AT 40
XI?PRINT Z(2)
 783

XI?PRINT Z(6)

UNDEFINED VALUE ACCESSED IN LINE 0

XI?Z(6)=445

XI?GOTO 30
 445


DONE
```

4.8 REM STATEMENT

Comments (on program use and/or construction for example) may be included in the program by means of REM statements.

SYNTAX

0013 REM THIS IS AN EXAMPLE OF A REM STATEMENT

FEATURES

1. REM statements may be included anywhere in a program.

2. Although REM statements are non-executable, it is legal to GOTO or GOSUB to them. Execution control passes to the first executable statement following the REM.

3. REM statements can be used to segment visually, different sections of lengthy programs. For example,

   0013 REM: PRINTING PHASE**********

   would be a very easy line in the program to find while scanning visually.

4.9 CHAIN STATEMENT

On occasion it is desirable to link several related programs together, or a single program may be too large to fit in main memory all at once. (The maximum amount of main memory available to the user is 10,112 words, and the amount of main memory used at any time may be determined by means of the LENgth command.) The CHAIN statement links one program to another by causing the first program to call for and execute its own replacement. The syntax is,

0013 CHAIN *program,line*

FEATURES

1. *Program* is the name of the replacement program. It may be a string constant, enclosed in quotes, or a string variable, subscripted or not. $ symbols preceding *program* are treated as they are in the GET command, and CHAINing to "$$$$" logs the user off the system. Shared programs may be CHAINed to as follows:

   0013 CHAIN *#account,program,line*

   where *#account,program* is treated as the full name of the program.

2. *Line* is optional, and specifies the line in the replacement program where execution will begin. If *line* is omitted then execution will begin at the first line in the replacement program. *Line* may be a numeric constant, variable or expression.

When the system encounters a CHAIN statement, error messages are generated if the replacement program is not in the designated library or if the program is too big to execute. Compile and execution error messages are given as for the RUN command.

After a successful CHAIN, the previous program will have been overwritten by the replacement program. (This is easily verified by LISting the current program both before and after execution.)

4.10 COM STATEMENT

When programs are CHAINed it is useful to be able to pass variables and files between the program without having respectively to re-define and re-link them. The COM statement permits this to be done conveniently. The syntax is,

0013 COM *variable,variable(dimension),file,file(buffer)*

FEATURES

1. *Variable* may be any string or numeric variable.
2. *Dimension* is used in COM statements as it is in DIM statements, except that all strings must have explicit dimension, even if the dimension is 1 as in DIM A$ (1).
3. *Buffer* is used in COM statements as it is in DIM statements.
4. For a file or a variable to be common to both the originating and the replacement programs, there must be a COM statement for it, both in the originating and in the replacement programs.

COM statements must be the first (lowest numbered) statements in both programs, and they must be contiguous. COM statements which appear later in the program are treated by the system as DIM statements. Variable names and file names which appear in COM statements must not appear elsewhere in the same program in DIM statements.

The buffer sizes of files, and the dimensions and data types of variables must match in both programs, but the actual names of the variables may differ. All items which appear in COM statements are passed one by one through the common area up to the point where the dimensions, buffer sizes or data types do not match.

Items may be passed through the common area only between contiguous programs. Two programs (PROG1 and PROG2 for example), are contiguous if one CHAINs to the other. The variable names in the three example programs illustrated below are identical from program to program. This is primarily for ease in understanding the explanation which follows the examples. The variables could just as well have had different names, so long as their data types and dimensions matched from program to program.

```
PROG1                              PROG2                              PROG3
0001 COM A$(254),B(2,3),#1(5)      0010 COM A$(254),B(2,3),#1(5)      0001 COM A$(254),B(2,3),#1(5)
0002 COM C(17),D$(15)              0020 COM C(17),#2(3),D$(15)        0002 COM C(17),#2(3),E(4,4)
   .                                  .                              0003 COM D$(15)
   .                                  .                                 .
   .                                  .                                 .
0100 CHAIN "PROG2"                 0750 CHAIN "PROG3"                   .
9999 END                           0760 END                           0100 END
```

PROG1 and PROG2 are contiguous, as are PROG2 and PROG3. PROG1 and PROG3 are not contiguous. The variables and files which are common to all three programs are, A$(254), B(2,3), #1(5) and C(17). File #2(3) is common to PROG2 and PROG3, and D$(15) is not common between any two of the three programs. (Its position in the COM lists is inconsistent.)

Variables and files in COM statements should be defined and linked, respectively in the program where they are first declared to be in COMmon.

4.11 APPend COMMAND

During program development it is often convenient to write large or complex programs in sections, save the sections separately, and then bring the sections together to form the whole.

Alternatively, the user may wish to include in his own program part or all of a program from another library. The APPend command provides this ability.

The APPend command retrieves and appends the referenced program to the program currently in the user's work area. The syntax is,

A. APP-*program*

B. APP-# *account, program*

FEATURES

1. *Program* is the name of the program to be appended and *account* (Syntax b.) is the account in whose library the (shared) program resides. If used, *account* must be preceded by a # symbol and followed by a comma.
2. The first statement number of the referenced program must be higher than the last statement number of the program in the user's work area, otherwise the system will respond:

   SEQUENCE NUMBER OVERLAP

   and the command will be ignored.
3. The shared list (if any) of the referenced program is replaced by the shared list (if any) of the program in the user's work area.

4. The user is cautioned that if the referenced program is RUN ONLY, then the (now combined) current and appended programs will be RUN ONLY and not available to LIST, PUNCH, XPUNCH, DELETE, RENumber or any program editing command.

To append a program from the System public library, the syntax is,

APP-$$$ *program*

To append a program from the Master library to which the current user has access the syntax is,

APP-$$ *program*

To append a program from the Group library to which the current user has access, the syntax is,

APP-$ *program*

Since APPend combine two programs into one program, there is no need to use the COM statement for variables and files referenced in the programs.

Chapter 5

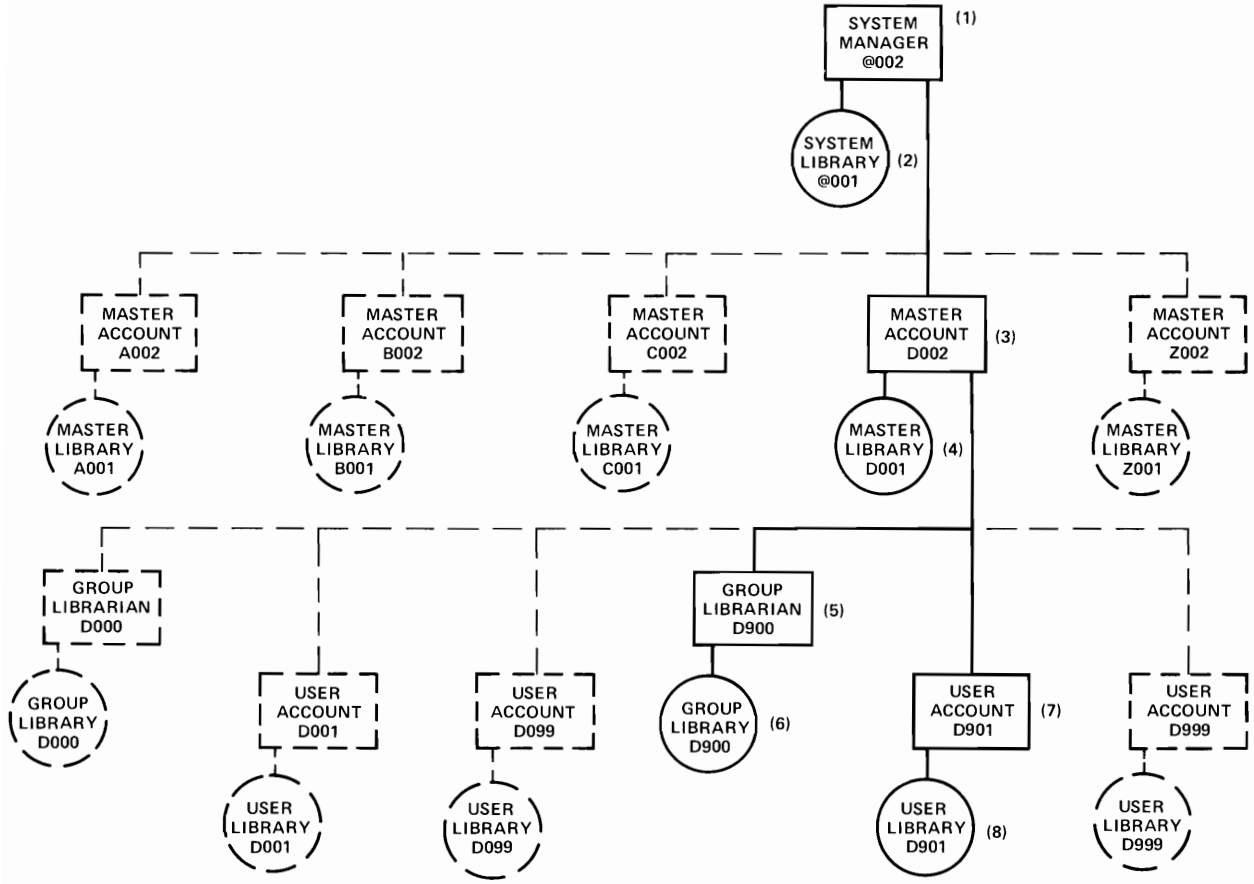
LIBRARIES

5.1 THE LIBRARY HIERARCHY

Account numbers in the BTI system are arranged in hierarchical structure. All account numbers in the system have three digits with a non-numeric prefix.

In the following discussion, the numbers in parentheses refer to the similarly numbered areas on the Library Hierarchy Chart. The chart illustrates the relationship between the various levels of the hierarchy. In descending order the level names are,

System Manager (1)
    System Library (2)
Master Account (3)
    Master Library (4)
Group Librarian (5) and User Account (7)
    Group Library (6) and User Library (8)



Library Hierarchy Chart

5.1.1 Special Purpose Accounts

Numbers prefixed by the @ ("AT") symbol have special purposes in the system. The @002 account (1) for example, belongs to the System Manager, and in the @001 account (2) resides the system public library. All users reference the files and programs in this library by using the file or program name preceded by three $ symbols. For example, the command GET-$$$EXAMPLE would cause the system to search the @001 library for a program named EXAMPLE. Regardless of which user types the command (this includes the @002 and @001 accounts) the system will search the @001 library if the file or program name is preceded by three $ symbols.

Next below the @002 level in the hierarchy are the '002 Master Accounts (3). The apostrophe denotes any letter of the alphabet, and there are 26 Master Accounts available in the system. For any Master Account, the associated Master Library resides in the '001 account (4). For instance, the library for the A002 Master Account resides in account A001 and is available to all users whose account numbers are prefixed by the letter A. Master Libraries are referenced by using two $ symbols preceding the file or program name. If user A367 types GET-$$EXAMPLE the system will search the A001 library for a program named EXAMPLE, but if user M403 types the same command (GET-$$EXAMPLE), the system will search the M001 library for the program named EXAMPLE.

5.1.2 User Accounts

User Accounts are next below Master Accounts in the hierarchy. The Group Librarians (5) are special-purpose User Accounts whose responsibility is to maintain their respective Group Libraries (6). Group Libraries are distinguished from other User Account Libraries by the fact that their account numbers have zeros as their two right-most digits, such as C200 or N900. Programs and files in Group Libraries are referenced by a single $ symbol preceding the name, and are available to all User Accounts whose letter prefix and left-most digit match those of the Group Library Account. For example, if account T289 types GET-$EXAMPLE the system will search the T200 Group Library for a program named EXAMPLE. If account B461 types the same command (GET-$EXAMPLE) then the system will search the B400 Group Library for the program named EXAMPLE.

Each User Account (7) has, in addition to its access to the Group, Master and System Libraries, a library of files and programs which belong to that account. The files and programs in a User Library (8) may have been created and modified by that user, or they may be copies of files or programs which are stored elsewhere in the system. These files and programs may be shared with other users at their owner's discretion, or they may be kept proprietary. A user references the files and programs in his private library by name, and without $ symbols. For example, if user D624 types

GET-EXAMPLE then the system will search User Library D624 for a program named EXAMPLE. Typical library accesses follow:

Command	GET-$$$EXAMPLE		
Account	G204	F437	T782
Library Access	@001	@001	@001
Command	GET-$$EXAMPLE		
Account	G204	F437	T782
Library Access	G001	F001	T001
Command	GET-$EXAMPLE		
Account	G204	F437	T782
Library Access	G200	F400	T700
Command	GET-EXAMPLE		
Account	G204	G437	T782
Library Access	G204	G437	T782

5.2 LIBRARY MANAGEMENT

The major elements of library management are:

1. adding files and programs to the library
2. erasing files and programs from the library
3. controlling access to files and programs in the library

The management of library files is discussed in detail in Chapter 6 of this Manual.

5.2.1 Adding Programs to the Library

A copy of the program in the user's work area may be added to the user's library by the SAVe command. The syntax is,

A. SAV-*programname*

B. SAV

FEATURES

1. Programs may only be saved in the user's own library.
2. The SAVe command decompiles the program currently in the user's work area, and any common values are lost.
3. *Programname* in Syntax a. establishes the name of the program as it will appear in the user's library catalog.
4. If an attempt is made to save a program with the same name as a file or program already in the library the error message will be printed:

   DUPLICATE ENTRY

5. *Programname* may be omitted as in Syntax b if the program already has a name, e.g.,

   a. if the program has already been named by the NAMe command, i.e.,

      NAM-*programname*

   b. if the program has been retrieved from a library (the user's own or a public library) and provided that a file or program of the same name does not already exist in the user's library.

5.2.2 Displaying the Contents of the User's Library

The CATalog command gives a full listing and description of each item in the user's library. The syntax is,

A. CAT

B. CAT-*m*

C. CAT-*m,n*

FEATURES

1. Syntax a causes the entire contents of the user's library to be displayed. For example,

```
CAT

 S/N      NAME        CODE   LEN     SECT.  SAVED    ACCESS    USE

ØØØ82    PK/MODE       P    ØØ32Ø    ØØØ3  188/75   188/75   ØØØØØ
ØØØ8Ø    EXAMPLE       P    ØØ249    ØØØ3  182/75   182/75   ØØØØØ
ØØØ79    TESTFILE      F    ØØØ1Ø    ØØØ11 178/75   182/75   ØØØ12

  TOTAL STORAGE = ØØØ17 SECTORS
```

2. Syntax b specifies the highest (most recent) serial number to be displayed.
3. Syntax c specifies the highest (most recent) and the lowest (oldest) serial number entries to be displayed.

The CATalog command is fully described in Appendix A.

The user's library catalog may also be output to the terminal or to a file by the System public library utility program named CATALOG. See Appendix E for details.

5.2.3 Renaming Catalog Entries

The name of a file or program in the user's library may be changed by the REDesignate command. The syntax is,

RED-*oldname,newname*

FEATURES

1. The Use Counter for the entry will be reset to zero, and the Date Accessed will be set to the date of the REDesignate command execution.
2. The entry's Serial Number and Date Saved will remain unchanged.
3. REDesignate has no effect on the copy of the program in the user's work area.

The REDesignate command is particularly useful when storing successively modified versions of the same program. For example,

```
(1)
CAT

S/N        NAME         CODE     LEN    SECT. SAVED  ACCESS   USE

ØØ122     TESTPROG      P       ØØØ36  ØØØ1  Ø61/76 Ø61/76  ØØØØØ
ØØ121     TESTPROG4     P       ØØØ36  ØØØ1  Ø5Ø/76 Ø61/76  ØØØØ6
ØØ12Ø     TESTPROG3     P       ØØØ36  ØØØ1  Ø49/76 Ø5Ø/76  ØØØØ2
ØØ12Ø     TESTPROG2     P

ABORT
(2)
GET-TESTPROG

(3)
LIS
TESTPROG

ØØ1Ø     DIM A$(10)
ØØ2Ø     INPUT A$(1:5)
ØØ3Ø     A$(7:10) = "1976"
ØØ4Ø     PRINT Z$
ØØ5Ø     END

(4)
2Ø       INPUT A$(1:6)
4Ø       PRINT A$

(5)
RED-TESTPROG,TESTPROG5

(6)
CAT

S/N      NAME         CODE    LEN   SECT. SAVED   ACCESS   USE

ØØ122  TESTPROG5      P     ØØØ36 ØØØ1  Ø61/76  Ø64/76  ØØØØØ
ØØ121  TESTPROG4      P     ØØØ36 ØØØ1  Ø5Ø/76  Ø61/76  ØØØØ6
ØØØ2Ø  TESTPROG3      P     ØØØ36 ØØØ1  Ø49/76  Ø5Ø/76  ØØØØ2

ABORT
```

```
(7)
SAV-TESTPROG

(8)
CAT

S/N      NAME         CODE    LEN.  SECT.  SAVED    ACCESS    USE

ØØ123    TESTPROG      P      ØØØ36 ØØØ1   Ø64/76   Ø64/76   ØØØØØ
ØØ122    TESTPROG5     P      ØØØ36 ØØØ1   Ø61/76   Ø64/76   ØØØØØ
ØØ121    TESTPROG4     P      ØØØ36 ØØØ1   Ø5Ø/76   Ø61/76   ØØØØ6

ABORT
```

Step 2 brings a copy of the most recent version of the program into the user's work area

Step 3 displays the program code

Step 4 modifies lines 20 and 40 of the program

Step 5 changes the name of the most recent *saved* version of the program from TESTPROG to TESTPROG5

Step 7 saves the (now) most recent version of the program into the user's library

5.2.4 Indexing Libraries

A short form of the contents of a library is displayed by the INDex command. The syntax is,

A.	IND	displays the user's own library
B.	IND-$$$	displays the System public library
C.	IND-$$	displays the associated Master library
D.	IND-$	displays the associated Group library

FEATURES

1. The names of programs and files in the referenced library are displayed alphabetically. File names are preceded by a # symbol.
2. For example,

```
IND-$$$

IND-$$$
INDEX OF: @001

 #2100MAC     #24K/AMPEX    300DEMO      300S         #32K/DIABLO
  32PORTLOG    6SQUARES     7200-PLOT1   7200-PLOT2    7200-PLOT3
  A2100        A2100PUN     ACCOUNT      ACCTFIND      ACCTLIST
  ACCTTITLE    ADD

ABORT
```

Files and programs may be veiled from display by the INDex command by the HIDe command. The syntax is,

A. HID-*file*

B. HID-*program*

The SEE command unveils hidden programs. The syntax is,

A. SEE-*file*

B. SEE-*program*

For example,

```
HEL-A123,

READY
IND

INDEX OF: A123
  BOOKS        DISCOUNT    EVERYONE    EXAMP       #FILE2      HELLO
  PER@ØØ2      PERMASTER   ROUNDOFF    #TESTFILE   TESTPROG    WX23

DONE
HID-TESTFILE
HID-EVERYONE
IND

INDEX OF: A123
  BOOKS        DISCOUNT    EXAMP       #FILE2       HELLO      PER@ØØ2
  PERMASTER    ROUNDOFF    TESTPROG     WX23
SEE-TESTFILE
IND

INDEX OF: A123
  BOOKS        DISCOUNT     EXAMP      #FILE2       HELLO      PER@ØØ2
  PERMASTER    ROUNDOFF    #TESTFILE    TESTPROG    WX23
```

5.2.5 Deleting Programs from the User's Library

Programs are deleted from the user's library by the KIL1 command. The syntax is,

KIL-*program*

The KIL1 command has no effect on the program currently in the user's work area.

5.2.6 Other Library Management Activities

Programs and files in the user's account library may be screened by a master account and/or by the System Manager from erasure and modification. (See Appendix A, the CATalog command.) The user may render programs in his own library unabortable by using the non-abort command. The syntax is:

NON-*program*

To neutralize NON-programs the non-abort status of a program the ABORT command is used. The syntax is:

ABO-*program*

The user is cautioned that if an unabortable program is screened by a higher-level account, only that account or a higher one can then neutralize the unabortable status of the program.

5.3 SHARED PROGRAMS

Programs, like files may be shared with as many as 118 different account numbers other than the account in whose library the programs reside. The SHAre command is used, and the syntax is,

SHA-*program,account*

FEATURES

1. *Program* is the name of the program to be shared and *account* is the name of the account with which program is to be shared.
2. Programs may be shared with public libraries and thus with all accounts which have access to those libraries. When a program is shared

   SHA-*program*,@001

   the System public library, it is available to all system accounts. When it is shared

   SHA-*program*,'001

   with a Master library, where ' stands for a specific letter of the alphabet, the program is available to accounts in that letter group. When a program is shared

   SHA-*program*,'x00

   with a Group library, where ' stands for a specific letter of the alphabet, and x for a specific digit between 0 and 9, the program is available to all accounts whose letter and left-most digit match those of the designated Group library account.
3. To GET a shared program from a "foreign" library, the syntax is,

   GET-#*account,program*

4. To CHAIN to a shared program in a foreign library, the syntax is,

   0013 CHAIN "#*account,program*"

5. To APPend a shared program from a foreign library to the program currently in the work area, the syntax is,

APP-#*account,program*

Each shared program carries with it a list of accounts with which it is shared. This list may be displayed by the System public library utility program named SHAREDLIST.

A shared program retains its original shared list under the following conditions:

A. when SAVed into a different account library
   - with its original name
   - when re-NAMed and then SAVed
   - when SAVed and then REDesignated

B. when re-NAMed and then saved into its originating library

C. when REDesignated in its originating library

D. when CHAINed to or from by its owning account or by an account with which it is shared

E. when RENumbered by its owning account or by an account with which it is shared

A shared program loses its shared list completely when APPended to another program, either in its originating account library, or in the library of an account with which it is shared. When APPended, a shared program's shared list is replace by the shared list (if any) of the program to which it is APPended.

The shared list of a program may be revoked in whole or in part by any account into whose library the program is saved, by the UNShare command. The syntax is,

A. UNS-*program*

revokes the entire shared list, and

B. UNS-*program, account*

revokes share privileges to the named account for the named program.

Chapter 6

FILES

A file is a collection of data, structured to conform to a user's specific needs. While they exist independently of BASIC programs, the data in the file are generally stored and retrieved by means of BASIC programs. Since files exist independently of user programs, the data in a file may be accessed and modified by more than one user at a time, under BASIC program control. The BASIC-X file subsystem provides special non-interfering file sharing features. These are described in Section 6.4.

6.1 STRUCTURE OF BASIC-X FILES

A file may contain string or numeric data or a combination of the two types. Each file is divided into a series of records, and a single file may have from 1 to 9502 records. Up to 63 separate files may be linked (i.e., referenced in a single program) by a given program at one time, and linked files may be freely passed in common between chaining programs. The number of file records required in a specific file depends (a) on how much storage space is required by the actual data in the file, and (b) how the file data are structured.

Each file record can hold as many as 128 words or 2048 bits of data. (One word is 16 bits.) Numbers are stored in floating point or "E" notation and each number requires 2 words or 32 bits of storage. Thus, one file record can store as many as 64 numbers.

Character strings require a half-word of storage per character, plus overhead. The formula for calculating string storage is 1 + INT[(N+1)/2] where N is the number of characters in the string. Stated in words, the formula says that each string requires one word for a string identifier, plus one word for every two characters in the string, plus a full word for an unpaired character, should one exist. Thus, a single file record can accommodate a string of up to 254 characters. This is proveable as follows:

$$1 + INT[(254 + 1)/2] = 128$$

We can solve for the number of words required to store a 10-character string in the same way:

$$1 + INT[(10 + 1)/2] = 6$$

Since a single file record can store 128 words of data, we could store up to INT(128/6) = 21 strings of 10 characters each, in one record. (As is demonstrated by this example, the overhead for each string is constant, plus or minus one word, so the larger the string, the less overhead is required to store it, relative to the length of the string itself.)

A file then, can store up to 600,000 floating point numbers or approximately 2.4 million characters, or a combination thereof.

Data are packed toward the beginning of each file record, and each item of data must be wholly contained within a record. That is, a string of 20 characters cannot be stored with 10 characters in record N and the remaining characters in record N+1. If an attempt is made to store data in such a way as to ignore this restriction, the user will either encounter an error, or the entire string will be stored in record N+1, depending on how the file is being used, i.e., as a random or serial access device. In either case, both time and file storage can be wasted if the user does not consider the construction of the data as it relates to the capacity of individual file records. Since a file record can hold a string of up to 254 characters in length, data can usually be constructed in such a way as to waste the minimum amount of space in each record.

The remainder of this chapter is devoted to the three major aspects of file use. They are,

creating/erasing/linking files
accessing files
sharing files

6.2 CREATING/ERASING/LINKING FILES

6.2.1 Creating and Erasing Files

Files are created either by the OPEn command or by File Mode 7. Files are erased either by the CLOse command or by File Mode 8. (File Modes are discussed later in this chapter.) The syntax for the OPEn command is,

OPEn-*filename*,*size*

FEATURES

1. *Filename* is the name of the file to be created. It may be any combination of up to 10 characters except that the following are disallowed:

   commas
   quotation marks
   non-printing characters (control characters)
   embedded blank spaces
   lower case letters
   leading # and $ symbols

2. *Size* is the number of records the file will contain. It is an integer in the range 1 to 9502 with the following restrictions:

   there must be sufficient space on the storage medium to accommodate a file of the specified size

   the user's storage limit must not be exceeded by creating a file of the specified size

For example, the command

OPE-FILEA,150

creates a file named FILEA which is 150 records long.

The syntax for the CLOse command is,

CLOse-*filename*

FEATURES

1. *Filename* is the name of the file to be closed.
2. Each record in filename is overwritten with End-Of-File marks.
3. The file is removed from the user's library.
4. The system types CLOSED.
5. Large files may require up to 5 minutes to be erased.
6. A file may be erased only if it is not linked, and not made permanent by a Master Account or the System Manager.

6.2.2 Linking Files

The FILE statement is used to create, erase and link files and the syntax is,

0013 FILE #*number,mode;name*

FEATURES

1. The # symbol must always precede *number.*
2. *Number* is an integer constant, variable or expression.
3. *Name* is the name of the file and is a string constant enclosed in quotes, or a string variable which may or may not be subscripted.
4. *Mode* is an integer constant, variable or expression which specifies the action to be taken, i.e., to create, erase or link the file.

The syntax for the FILE statement to create files is,

0013 FILE #*size*, 7;*filename*

FEATURES

1. *Size* is a positive integer constant, variable or expression which evaluates to <9503. It specifies the number of records the file is to contain.
2. The integer 7 designates the file create mode and is the only mode which creates a file. Therefore, if a variable or expression is used, it must evaluate to the quantity 7.
3. *Filename* is the name of the file to be created.

The syntax for the FILE statement to erase files is,

0013 FILE #*number*,8;*filename*

FEATURES

1. *Number* is a positive integer constant, variable or expression which evaluates to less than 32767. Its purpose is to maintain a consistent syntax for FILE statements. Even though number has no direct use in erasing the named file, it is evaluated by the system. This means that if a variable is used to specify *number*, the variable must have been previously defined in the program.
2. The integer 8 designates the file erase mode and is the only mode which erases a file. Therefore, if a variable or expression is used, it must evaluate to the quantity 8.
3. *Filename* is the name of the file to be erased.
4. A file may be erased only if it is not currently linked. To unlink a file, a statement of the following form may be executed:

   0013 FILE #*number*, 3;" "

   where *number* is the number of the currently linked file which is to be unlinked. The general use of Mode 3 is discussed in Section 6.5.5. The filename in this case may be any name other than that of a file currently extant in the user's library. The null (" ") form is suggested merely as a convenience. For example, to unlink file number 10,

   0013 FILE #10,3;" "

In order to be available for data storage and retrieval, an existing file must be linked. The syntax for the FILE statement to link files is,

0013 FILE #*filenumber*,*mode*;*filename*

FEATURES

1. The FILE statement, using mode 1 through mode 6 links the named file to the current program.
2. *Filenumber* is an integer constant, variable or expression which evaluates to a number in the range 1 to 63.
3. *Mode* is an integer constant, variable or expression in the range 1 through 6. The *mode* is optional for linking files and if not included, the system sets the mode to 1 (READ ONLY). File modes are discussed in detail in Section 6.5.5. The syntax without *mode* is,

   0013 FILE #*filenumber*;*filename*

4. *Filename* is the name of a previously created file to be accessed. It may be a string constant enclosed in quotes, or a string variable, subscripted or not.

One or more FILE statements must be executed for each file to be accessed in the program. To access a file residing in a library other than the user's own, the FILE statement syntax is,

0013 FILE *#filenumber,mode;#account,filename*

The file must previously have been shared with the current user. The # symbol must precede the account, and *account* is the name of the account which owns the named file, that is the account in whose library the file resides. For purposes of linking files, the entire expression, *# account,filename* is considered to be the name of the file, and may be specified either as a string constant enclosed in quotes, or as a string variable, subscripted or not. For example,

0013 FILE #1,1;"#M050,TESTFILE"

Files which belong to Group, Master or System accounts are shared on a READ ONLY basis by virtue of belonging to the public account. They may be linked as follows:

Group library files	0013 FILE *#filenumber,mode;$filename*
Master library files	0013 FILE *#filenumber,mode;$$filename*
System library files	0013 FILE *#filenumber,mode;$$$filename*

6.3 ACCESSING FILES

There are two classic types of file data storage; random and serial. In a random access file, the data to be stored are organized into sets and each set is stored in a separate file record or contiguous group of records. Retrieval of data stored in this manner is straightforward, since each data set is directly related to a specific file record number.

In a serial access file, each data set is stored on the heels of its predecessor without regard to file record numbers. Data retrieval is done by stepping through data sets until the desired set is found. This method is typically used to store very small and very large data sets since there is less chance of wasted space at the ends of records (see Section 6.1). Even in a serially accessed file, each item of data must be wholly contained within a given record, however.

In most applications, some combination of random and serial access is used.

6.3.1 Reading Data From Files

The syntax for random file READ statements is,

A. 0013 READ *#filenumber,recordnumber;variable,variable,. . .variable*

and the syntax for serial file READ statements is,

B. 0013 READ *#filenumber;variable,variable,. . .variable*

FEATURES

1. The # symbol must always precede *filenumber.*
2. *Filenumber* is an integer constant, variable or expression which evaluates to the number with which the file is linked, and must always be specified. For example,

```
0001 FILE #1,1;"FILEA"
0002 FILE #2,1;"FILEB"
0003 FILE #3,1;"FILEC"
```

now there are three files linked. File number 1 is named FILEA, file number 2 is named FILEB, and file number 3 is FILEC. All three files must be in the current user's library, that is they must all have been previously created by one of the methods described in Section 6.2.2. To read a 6-character string to be named A$ from the 10th record of FILEC,

```
0004 DIM A$(6)
0005 READ #3,10;A$
```

3. In a random file read (Syntax A), a *recordnumber* is specified. It is specifying a record number that makes the statement a random file read. (If *recordnumber* is omitted as in Syntax B, the statement is a serial file read.) *Recordnumber* is an integer constant, variable or expression which evaluates to the number of the file record from which the variables are to be read. *Recordnumber* may not be larger than the size of the file.
4. *Variable* will be the name of the string or number to be read from the file. It must match the data in the file in type and dimension (see Section 6.3.2). In a random file read (Syntax A) all of the variable(s) to be read must be contained in the specified record. In a serial file read, the variables must be contiguous in the file, but may be stored in more than one (contiguous) record.
5. The file READ statement reads strings of any length up to 254 characters and/or numbers, one at a time. For example,

```
0013 DIM Z$(20)
0014 READ #1,10;Z$,B,C
```

reads from record 10 of file 1, a string no longer than 20 characters to be named Z$, and two numbers to be named B and C, respectively.

The MAT READ statement is used to read numeric arrays. For random file reads, the syntax is,

0013 MAT READ #*filenumber,recordnumber;array,array,. . .array*

and for serial file reads, the syntax is,

0013 MAT READ #*filenumber;array,array,. . .array*

FEATURES

1. The dimension(s) of *array* may be specified either in a DIM statement or in the MAT READ statement. For example, either

```
(1) 0001  DIM W(23),X(16),Y(1),Z(3,3)
    0002  MAT READ #4,4;W,X,Y,Z
```

or

```
(2) 0003  MAT READ #2;N(4,2),R(10),T(6)
```

is correct. (The dimensioning of arrays for files follows the same rules that apply to all other array use. Specifically, if the array is to be larger than 10 or 10,10 then it must previously have been dimensioned in a COM or DIM statement. See Section 2.2.)

2. In a random file MAT READ such as example (1) above, the total quantity of numbers contained in the specified record must be greater than or equal to the quantity of numbers to be read by the MAT READ statement. In example (1) above, the quantity of numbers to be read is 49.

```
W(23)   = read 23 numbers
X(16)   = read 16 numbers
Y(1)    = read  1 number
Z(3,3)  = read  9 numbers

          read 49 numbers
```

If record 4 of file 4 contains fewer than 49 numbers, then an End-Of-Record error will be generated. An End-Of-Record error will also be generated by any random file READ statement which attempts to read more than 64 numbers, since there can be no more than 64 numbers in any one file record. A MAT READ or MAT PRINT of >64 numbers is done by using serial access to the file, that is by omitting the file record number.

6.3.2 Writing Data To Files

The syntax for random file PRINT statements is,

A. 0013 PRINT #*filenumber,recordnumber;item,item,. . .item*

and the syntax for serial file PRINT statements is,

B. 0013 PRINT #*filenumber;item,item,. . .item*

FEATURES

1. The # symbol must precede *filenumber.*
2. *Filenumber* is an integer constant, variable or expression which evaluates to the number with which the file was linked in a previous FILE statement.

3. In a random file print, (Syntax A) the *recordnumber* must be specified. It is an integer constant, variable or expression which evaluates to the number of the file record into which the item(s) are to be written.

4. *Item* is the string constant or variable, or the numeric constant, variable or expression whose value is to be written to the file. In a random file print (Syntax A) there must be sufficient space remaining in the specified record to hold the item(s), and in no case can the total storage requirement of the data to be printed equal more than 128 words, whereas in a serial file print (Syntax B) these restrictions do not apply.

5. The file PRINT statement prints strings of any length up to 254 characters, and/or numbers, one at a time. For example,

```
0013 DIM C$(17)
0014 PRINT #1,6;C$,D,E
```

prints in record 6 of file 1 a 17-character string, C$ and two numbers, D and E respectively. Note that it is the *logical* length of C$ that will be written to the file and not the *dimensioned* length. If C$ = "13 CHARACTERS" then 13 characters will be written to the file, and a string variable of dimension 13 can read the string data. In this connection, it is worth noting further that the names of variables specified as items in a file print statement bear no relation to the names of variables which might later be used to read the items.

The MAT PRINT statement is used to write numeric arrays to a file. For random file MAT PRINTS the syntax is,

A. 0013 MAT PRINT *#filenumber,recordnumber;array,array,. . .array*

and for serial file MAT PRINTS the syntax is,

B. 0013 MAT PRINT *#filenumber;array,array,. . .array*

FEATURES

1. The dimensions of arrays to be printed cannot be specified in the file MAT PRINT statement, but must be defined previously in the program.

2. In a random file MAT PRINT, the total quantity of numbers to be printed must not be greater than 64. For example,

```
0011 DIM A(8,9)
0012 FOR I=1 TO 72
0013 A(I)=I
0014 NEXT I
0015 MAT PRINT #1,7;A
0016 END
```

will generate an End-Of-Record error in line 15 of the program, whereas the serial access version,

```
0015 MAT PRINT #1;A
```

will not.

When printing strings to a file, keep in mind that a subset of the string cannot be read from the file. For example, if a 15-character string is printed to a file, that string can only be read by a string variable whose dimension is greater than or equal to 15. The sequence

(1)
```
0001 DIM X$(15),B$(2)
0002 X$="ABCDEFGHIJKLMNO"
0003 FILE #1,2;"TEST"
0004 PRINT #1,1;X$
0005 READ #1,1;B$
0006 PRINT B$
0007 END
```

will produce a STRING OVERFLOW error in line 5. The inverse approach will work, however.

(2)
```
0001 DIM X$(15),B$(2)
0002 B$="YZ"
0003 FILE #1,2;"TEST"
0004 PRINT  #1,1;B$
0005 READ #1,1;X$
0006 PRINT LEN(X$)
0007 PRINT X$
0008 END
RUN
 2
YZ

DONE
```

Further note that if another string immediately follows the string to be read as X$ in example (2) above, that string will not be read as a part of X$, even if it would fit into the 15-character variable along with the 2-character string stored under the name B$.

Numbers printed to a file in array form may be read in subsets, including as single numbers. For example,

```
0001 DIM Z(3,4),A(5),B(2,2)
0002 MAT READ Z
0003 DATA 1,2,3,4,5,6,7,8,9,10,11,12
0004 FILE #1,2;"TEST"
0005 MAT PRINT #1,1;Z
0006 MAT READ  #1,1;A,B
0007 READ #1;D,E,F
0008 END
```

The value of the variables A, B, D, E and F after execution of line 7 are,

$$A = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix} \qquad B = \begin{bmatrix} 6 & 7 \\ 8 & 9 \end{bmatrix} \qquad D = 10 \qquad E = 11 \qquad F = 12$$

6.3.3 File Data Location

The physical End-Of-File is that record in a file beyond which data may not be stored. It is established by the user for each file in the user's private library when each file is created.

FEATURES

1. If K was the number in the *size* parameter when the file was created then the physical End-Of-File follows the Kth record.
2. The physical End-Of-File cannot be modified in any way by the user, once it is set.
3. An End-Of-File mark is a simulated physical End-Of-File which may be written to the file by the user.
4. End-Of-File marks are treated the same as the physical End-Of-File during file read operations, but may be overwritten in write operations.
5. An End-Of-Record mark is a simulated record boundary which follows the last data item in a record, and which may be written to the file by the user.
6. End-Of-Record marks are treated the same as record boundaries during file read operations, but may be overwritten in write operations.
7. There may be either an End-Of-File mark or an End-Of-Record mark in each record of a file, but not both.

A flag may be set in programs to detect and avoid impending errors that would be generated by trying to read or write beyond the end of the file. The syntax is,

0013 ON END #*filenumber* THEN *linenumber*

FEATURES

1. *Filenumber* is an integer constant, variable or expression which evaluates to the number of the referenced file. It must always be preceded by the # symbol.
2. *Linenumber* is the number of the line in the current program to which execution control will pass in the event that the End-Of-File is encountered by an attempted file operation in the current program.
3. The ON END flag for any file may be revoked as follows:

   0013 ON END #*filenumber* THEN 0

This will allow End-Of-File conditions to generate errors as though a flag had never been set for that file.

4. The ON END flag is revoked by chaining to a new program.

ON END statements must be preceded in the program by FILE statements so that the End-Of-File statement will reference an existing, linked file. Once the ON END flag is set for a given file it remains in effect until either it is explicitly changed in the current program, or the current program ends. The user is cautioned that since the ON END statement references a file number and not a file name, care should be taken to re-define the ON END flag if a new file is linked with a previously used file number for which an ON END flag was already set.

The ON END flag is set when the ON END statement is executed. Therefore, if a file READ or PRINT statement is executed for a given file earlier in the program than an ON END statement for that file is executed, any End-Of-File errors generated would not be handled by the ON END flag.

The ON END flag is triggered by End-Of-File marks in the same way as it is by the physical End-Of-File. During file write operations, End-Of-File marks are handled as data, in that they may be written to the file, overwritten by other data, or may overwrite other data.

It is frequently necessary to know the type of the next data in the file before attempting to read from or write to a file. The TYP function allows the next data type in the file to be determined without the risk of generating a terminal error. The syntax is,

A. 0013 A=TYP *(filenumber)*

B. 0013 A=TYP *(-filenumber)*

C. 0013 IF TYP*(filenumber)*=N THEN *linenumber*

D. 0013 GOTO TYP *(filenumber)* OF *linenumber,linenumber,. . .linenumber*

FEATURES

1. The minus sign preceding the file number is optional. If used, as in Syntax B then five values are possible in the result:

   0 = the referenced file is non-existent — (if the file is "linked" using Mode 4, 0 can mean either that the file is busy or non-existent)

   1 = the next data item is numeric

   2 = the next data item is string

   3 = the next item is an End-Of-File

   4 = the next item is an End-Of-Record

   The minus sign may also be used in Syntaxes A, C and D.

2. If a minus sign is omitted then one of only the first four values (0 through 3) is possible in the result, and End-Of-Record marks and record boundaries are ignored.

3. A newly linked file shows TYP(-*N*)=4 even if there is an End-Of-File mark or other data in the record.

During program execution, each file that is to be accessed by the program has a pointer as soon as the file is linked. In addition, each user who is accessing a file has a separate pointer to the file. Immediately following execution of a file statement, the pointer is set to the beginning of the file.

Both the file READ and file PRINT statements may be used in "itemless" form, that is where they have no variable or item parameter, respectively. The formats for these itemless statements and their uses are:

A. 0013 READ #F,R

positions the file pointer at the Rth record of file number F.

B. 0013 PRINT #F,R

positions the file pointer at the Rth record of file number F and writes an End-Of-Record mark in that record. The End-Of-Record mark replaces any and all data stored in the referenced record.

C. 0013 PRINT #F,R;END

writes an End-Of-File mark in the Rth record of file number F and positions the file pointer after the End-Of-File mark.

D. 0013 PRINT #F

writes the current buffer for file number F back to the disk. (See Section 6.4.2.)

Below is a graphic representation of the file pointer location following execution of typical file operations.

file name: FILE
file length: 5 records

Record #1 = "A 21 character string" — 12 words of storage
Record #2 = 4002 4713 84 17 467 853 — 12 words of storage
Records #3, 4 and 5, unspecified — 0 words of storage

EOR = End-Of-Record EOF = End-Of-File * = physical End-Of-File : = record boundary

ARROW INDICATES POINTER POSITION IN FILE AFTER STATEMENT EXECUTION

STATEMENT RUN	: #1	: #2	: #3	: #4	: #5 :
0010 FILE #1,2;"FILE"	:A 21 character string	EOR:4002 4713 84 17 467 853 EOR:			:EOF*
0020 READ #1,1	:A 21 character string	EOR:4002 4713 84 17 467 853 EOR:			:EOF*
0030 DIM A$(21)					
0040 READ #1;A$	:A 21 character string	EOR:4002 4713 84 17 467 853 EOR:			:EOF*
0050 READ #1;B	:A 21 character string	EOR:4002 4713 84 17 467 853 EOR:			:EOF*
0060 DIM C(3)					
0070 MAT READ #1;C	:A 21 character string	EOR:4002 4713 84 17 467 853 EOR:			:EOF*
0080 D=906					
0090 PRINT #1;D	:A 21 character string	EOR:4002 4713 84 17 906 EOR:			:EOF*
0100 READ #1,2	:A 21 character string	EOR:4002 4713 84 17 906 EOR:			:EOF*
0110 PRINT #1,3;END	:A 21 character string	EOR:4002 4713 84 17 906 EOR:	EOF:		:EOF*
0120 PRINT #1,4	:A 21 character string	EOR:4002 4713 84 17 906 EOR:	EOF:	EOR:	:EOF*
0130 FILE# 1,1;"FILE"	:A 21 character string	EOR:4002 4713 84 17 906 EOR:	EOF:	EOR:	:EOF*

6.4 COMMON FILES AND FILE BUFFERS

6.4.1 Common Files

Linked files may be declared to be in common between chaining programs. The COM statement is used for this purpose, and optionally to specify the size of a given file's buffer. To declare a file to be in common the syntax is,

A. 0013 COM *#filenumber*

and to specify the common file's buffer size the syntax is,

B. 0013 COM *#filenumber (buffer)*

FEATURES

1. *Filenumber* is an integer constant which is equal to the number with which the file is to be linked. *Filenumber* must be preceded by a # symbol.
2. *Buffer* (Syntax B) is an integer constant in the range 1 through 32 which specifies the size of the buffer for *filenumber* in terms of number of records. For example, the statement

   0013 COM A(6,3),Z$(254),#1(10),#2(4),#3

   declares the following to be in common:

   the numeric array variable A whose dimensions are 6 rows by 3 columns
   the string variable Z$ which may be up to 254 characters long
   the file number 1 with a buffer equal to 10 file records in length
   the file number 2 with a buffer equal to 4 file records in length
   the file number 3 with a buffer equal to 1 file record in length (the buffer for a given file is 1 record long if its size is not explicitly declared to be otherwise)
3. Like all COM statements, those specifying files must be the first statement(s) in both the current program and in the chained program.
4. Like variables in COM statements, files passed in common must be in corresponding positions in the COM statement and must have the same buffer sizes if buffer sizes are specified. (See Section 4.10.)

6.4.2 File Buffers

A file buffer is a space in the user's work area which is reserved to hold a copy of data being read from or written to the file. Each linked file has a separate buffer which is reserved when the file is initially linked. The actual file and its data remain stored on the disk. The purpose of file buffers is to minimize the number of disk accesses during program execution. This is because disk access is more time consuming than main memory access.

File buffer sizes may be declared in DIM statements as follows:

0013 DIM #*filenumber (buffer)*

FEATURES

1. *Filenumber* is an integer constant equal to the number with which the file is linked.
2. *Buffer* is a positive integer constant <33, which specifies the size of *filenumber*'s buffer in terms of number of records.
3. A file may not have its buffer size set (or implied) twice in the same program. This means that if the file is to be passed through the common area to a chaining program, its buffer size may not be declared in a DIM statement, either in the current program or in the chained program. This is true even if no buffer size is specified for the file in the COM statement.
4. An example of the DIM statement used to set file buffer sizes,

   0013 DIM A$(254),#1(25),D(2,3)

   declares the dimensions of the following:

   the string A$ to have up to 254 characters
   the file number 1 to have a buffer equal to 25 file records in length
   the array variable D to have 2 rows and 3 columns

When an operation on a file (a file READ or a file PRINT) "pushes" the pointer beyond the end of the buffer, the contents of the buffer are written back to the disk, and the record(s) containing the data referenced by the file operation are read into the buffer. In most applications, data sets are stored and retrieved in contiguous fashion. Therefore, the larger the file buffer, the fewer disk accesses are necessary, since a larger piece of the file can fit into the buffer at one time. The user is advised to exercise caution when specifying large file buffers. The available work area is 10,112 words, and each 1-record increment in buffer size uses an additional 128 words of main memory. Thus in general it is prudent to keep file buffers in the range 1 to 5.

When several users are sharing a file simultaneously, there exists the possibility that the buffer belonging to the user with READ/WRITE access will not have been written back to the disk in time for a READ ONLY access to make use of data written by the READ/WRITE user. (See Section 6.5.5, File Access Modes.) The probability increases in direct proportion to the size of the users' buffers. In order to strike the optimum balance between buffer sizes and buffer "dump frequencies", the following is provided as a means of writing (dumping) the buffer back to the disk under program control:

0013 PRINT #*filenumber*

The buffer is written back to disk automatically when a file is re-linked. This is of particular interest in simultaneous file sharing applications where the File Mode alternates frequently between Mode 5 (READ ONLY) and Mode 6 (READ/WRITE).

6.5 SHARED FILES

6.5.1 SHAre Command

Data stored in files may be shared among many users. In order for a file to be available to accounts other than the account in whose library the file resides, the file must be shared with the other accessing accounts. The SHAre command is used for this purpose and the syntax is:

SHA-*filename,account*,R

FEATURES

1. *Filename* is the name of the file to be shared.
2. *Account* is the name of the account with which *filename* is to be shared.
3. R is optional. If included, the file is shared on a READ ONLY basis with the named account. If R is omitted then the shared file access is READ/WRITE.
4. User files may be shared in the same way with all levels of public libraries. When a file is shared

   SHA-*filename*,@001

   with the System public library, it is available to all system accounts. When it is shared

   SHA-*filename*,'001

   with a Master public library, where ' stands for a specific letter of the alphabet, the file is available to all accounts within the designated letter group. When a file is shared

   SHA-*filename*,'x00

   with a Group public library, where ' stands for a specific letter of the alphabet and x for a specific digit between 0 and 9, the file is available to all accounts whose letter and left-most digit match those of the designated Group public library.

   A file may be shared simultaneously with a public library and with individual accounts. For example,

   SHA-FILEA,@001
   SHA-FILEA,B206
   SHA-FILEA,N483

   In this case, even if the @001's share privileges are revoked, the share privileges of accounts B206 and N483 remain. See the UNShare command for further information on revoking file sharing privileges.

6.5.2 Listing File Access

The System public library maintains a utility program named SHAREDLIST. The program lists, for the designated file, all accounts with which the file is shared. The file must either be owned by or

shared with the inquiring account. For example,

```
HEL-A1Ø1
READY
SHA-FILE1,A2Ø2
SHA-FILE1,A567,R
SHA-FILE2,A2Ø2,R
HEL-A2Ø2,
READY
SHA-FILE3,A444
SHA-FILE3,A555
```

The SHAREDLIST utility flags files shared READ ONLY with an asterisk.

```
GET-$$$SHAREDLIST
RUN
SHAREDLIST
NAME: #A1Ø1,FILE1
A202              *A567
NAME: #A1Ø1,FILE2
*A202
NAME: FILE3
A444              A555
NAME: FILE4
NO SUCH ENTRY
NAME:
DONE
```

Accounts with which a given file has been shared have the same priviliges as the owning account (unless access has been limited to READ ONLY by the R parameter in the SHAre command) with two exceptions: one, the non-owning account(s) cannot control share privileges to another account, and two, the non-owning account(s) cannot close the file. The account which owns the file retains sharing privileges, READ/WRITE privileges and closing privileges, unless the file has been protected by its Master account or the System Manager.

6.5.3 Copying Files

Data may be copied from one file to another by the COPy command. The syntax is:

COP-*#source account,source filename,#destination account,destination filename,*
first source record,last source record,first destination record

FEATURES

1. Only the underlined parameters are mandatory. For example, if account A123 executes

   COP-TESTFILE,MYFILE

   the whole of TESTFILE will be copied to MYFILE, both files in the library of account A123.
2. If the source account and/or destination account are specified, each must be preceded by a # symbol. A user may copy a file *from* a source account different from his own if the source file has been shared with the copying user. The user may copy *to* a destination file in an account different from his own if the destination file has been shared on a READ/WRITE basis with the copying user.
3. The *first source record*, *last source record* and *first destination record* parameters permit less than whole files to be copied. The user may specify a range of source file records to be copied, or may even copy a single record from the source file. For example,

   (1) COP-FILEA,FILEB,2,7,1

   (2) COP-FILEA,FILEB,3,3,10

   In example (1), records 2 through 7 of FILEA will be copied to FILEB, beginning in record 1 of FILEB. In example (2), record 3 of FILEA will be copied to record 10 of FILEB.
4. If the *first source record* parameter is specified, then both the *last source record* and the *first destination record* parameters must be specified.
5. The COPy command causes any data previously stored in the destination file to be overwritten with the specified data from the source file, and leaves the source file unaffected.
6. The destination file must be large enough to accommodate the specified portion of the source file.

6.5.4 UNShare Command

The UNShare command revokes share privileges to the named file for the named account. The syntax is:

UNS-*filename,account*

FEATURES

1. *Filename* is the file which the named *account* may no longer share.
2. To revoke all share privileges to a given file:

   UNS-*filename*

6.5.5 File Modes

During interactive and simultaneous file sharing, there could exist the possibility that a given file record would be updated simultaneously by two or more users, leading to inaccurate or garbled file data. BASIC-X, to preclude this possibility, implements file sharing on a non-interfering basis. This is, if one

user is writing to a file, BASIC-X locks out all other attempts to establish a WRITE link to the file. Other users have the option of having their requests for WRITE access queued or ignored.

File modes 1 through 6 are used to link files. The eight file modes and their uses are:

MODE	PURPOSE
1	READ ONLY link
2	READ/WRITE link
3	READ ONLY link (file de-linked if non-existent)
4	READ/WRITE link (file de-linked if non-existent or if a READ/WRITE link exists)
5	READ ONLY link (only for files already linked in another mode)
6	READ/WRITE link (only for files already linked in another mode — request is queued if a READ/WRITE link exists)
7	CREATE FILE
8	DELETE FILE

There are numerous circumstances which might prevail at the time a FILE statement is executed. For example, the file may reside in a Public library and be shared with all users READ ONLY. Or the file might be shared READ/WRITE with all users, opening the possibility of its being unavailable to any one user from time to time.

The following chart is provided to help the user understand and anticipate the behavior of the file modes under typical use conditions. It is by no means a complete list of every circumstance under which the user might execute a FILE statement, but is intended to assist in making full and efficient use of BASIC-X files.

In each case the referenced file is presumed to reside in the current user's library, to be shared with at least one other user on a READ/WRITE basis, and to have no Master account or System Manager protection. (See Appendix A, the CATalog command for a discussion of higher level account protection of files and programs.)

In the chart, an entry in caps represents an error message. The phrase "file de-linked" means that *any* file which was previously linked in the program with the same file number is now not linked, and that no new link to that file number has been established.

In every case where the file is successfully linked, the file pointer is set to the beginning of the file, and the file buffer (if any) is written back to disk.

	File not linked by any user	File linked by current user	File linked by current user and in a READ/ WRITE mode by another user	File nonexistent
Mode 1 (READ ONLY)	file linked	file re-linked	file re-linked	NO SUCH FILE
Mode 2 (READ/WRITE	file linked	file re-linked	BUSY/PROTECTED FILE	NO SUCH FILE
Mode 3 (READ ONLY)	file linked	file re-linked	file re-linked	file de-linked
Mode 4 (READ/WRITE)	file linked	file re-linked	file de-linked	file de-linked
Mode 5 (READ ONLY)	NON-EXISTENT FILE REFERENCED	file re-linked	file re-linked	NON-EXISTENT FILE REFERENCED
Mode 6 (READ/WRITE)	NON-EXISTENT FILE REFERENCED	file re-linked	program suspended until file available	NON-EXISTENT FILE REFERENCED
Mode 7 (FILE CREATE)	DUPLICATE ENTRY	DUPLICATE ENTRY	DUPLICATE ENTRY	file created
Mode 8 (FILE DELETE)	file deleted	BUSY/PROTECTED FILE	BUSY/PROTECTED FILE	NO SUCH FILE

As is demonstrated by the above chart, in order to link a file in Mode 5 or Mode 6, the file must currently be linked. In other words, while it is legal to go to Mode 5 from Mode 6 and vice versa, it is not legal to execute a FILE statement using Mode 5 or Mode 6 as the initial link for that file number.

Following is a list of the most common file error messages in alphabetical order by first word, and some of their typical causes.

BAD FILE NAME

1. File name exceeds 16 characters (Modes 1 through 6) or 10 characters (Modes 7 and 8).
2. File name includes blanks and/or disallowed characters.

BUSY/PROTECTED FILE

1. File Mode 2 executed for a file with a current READ/WRITE access.
2. File Mode 2 executed for a file with Master account and/or System Manager account protection. (See Appendix A, the CATalog command.)
3. File Mode 2 or 6 executed for a file in a "foreign" library which has been shared READ ONLY with the current user.

DUPLICATE ENTRY

File Mode 7 executed using the name of a file or program already in the current user's library.

NO SUCH FILE

1. File Mode 1, 2 or 8 executed for a file name which does not exist in the current user's library.
2. File Mode 1 or 2 executed for a file in a "foreign" library to which the current user has been granted no shared privileges.

NON-EXISTENT FILE REFERENCED

File Mode 5 or 6 executed for a file not previously linked in the current or chained program.

WRITE TRIED ON READ ONLY FILE

File PRINT statement executed with file linked in Mode 1, 3 or 5.

Of particular interest in this discussion are File Modes 3, 4, 5 and 6. Mode 3 (READ ONLY) and Mode 4 (READ/WRITE) are constructed to avoid terminal errors due to attempts to link a non-existent file or a file with an existing READ/WRITE access.

If the referenced file does not exist, then a FILE statement specifying Mode 3 and Mode 4 will de-link any file previously linked using the specified file number. The same is true if Mode 4 is used in an attempt to link a file with an existing READ/WRITE access. For example, assume that the file MYFILE has a current READ/WRITE link on another port. The success of the attempted link may be tested using the TYP function.

```
ØØ1Ø FILE #1,1;"MYFILE"
ØØ2Ø PRINT "TYPE OF FILE #1, MYFILE IS ";TYP(1)
ØØ3Ø FILE #1,4;"MYFILE"
ØØ4Ø PRINT "TYPE OF FILE #1, MYFILE IS ";TYP(1)
ØØ5Ø END

RUN

TYPE OF FILE #1, MYFILE IS  1
TYPE OF FILE #1, MYFILE IS  Ø

DONE
```

Attempts to read from or write to a de-linked file will generate terminal errors. For example, assume that MYFILE has a current READ/WRITE link on another port:

```
0010  FILE #1,1;"MYFILE"
0020  READ #1;A
0030  PRINT A
0040  FILE #1,4;"MYFILE"
0050  READ #1;A
0060  PRINT A
0070  END

RUN

87540

NON-EXISTENT FILE REFERENCED IN LINE 50

STOPPED AT 60
XI?
```

The above program assumes the existence of the file. If neither the existence of the file nor its availability for WRITE access is established, then the program might be written as follows:

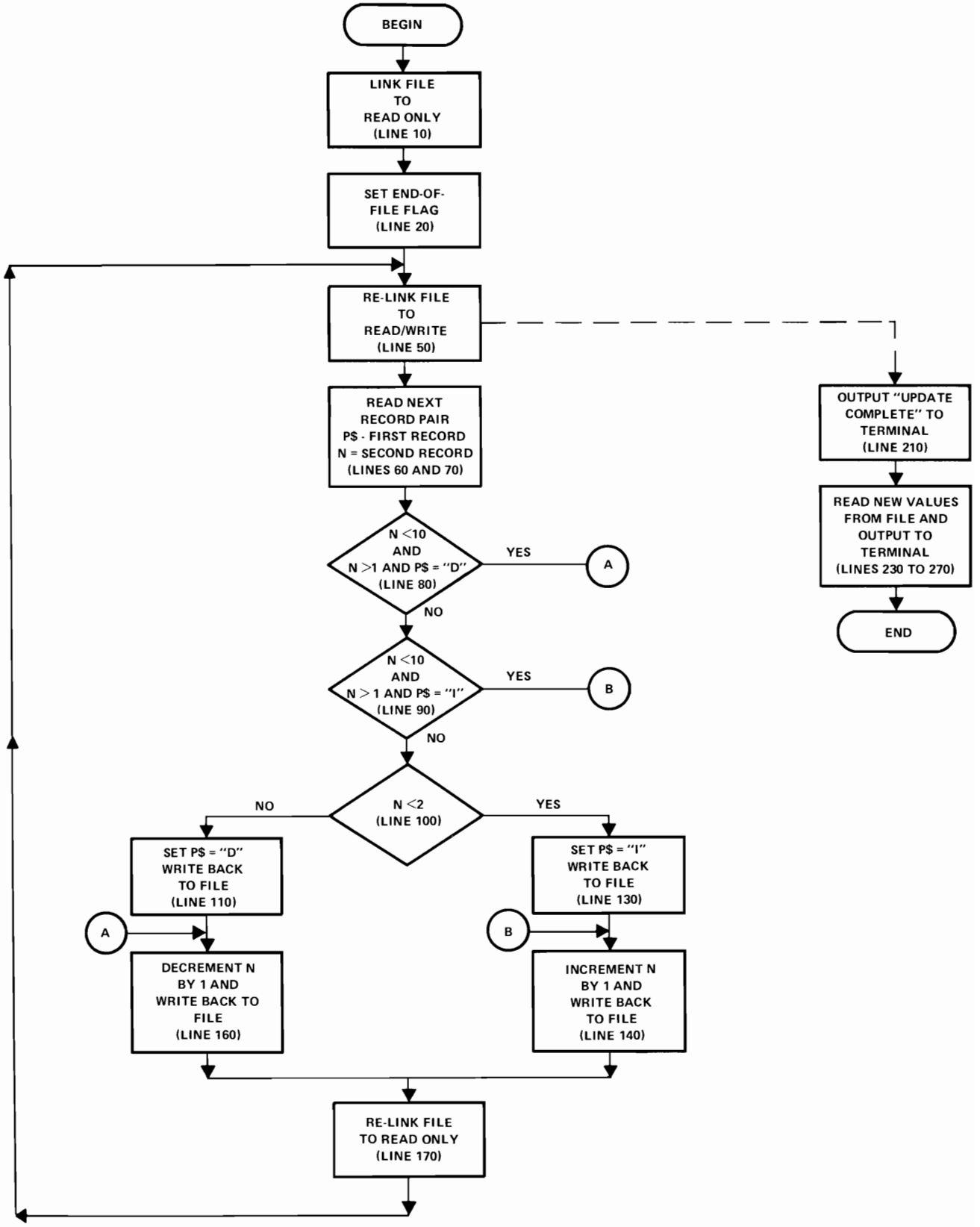
```
0001  ON ERROR THEN 100
0002  DIM F$(16)
0003  INPUT "FILE NAME: ",F$
0004 FILE #1,3;F$
0005  IF TYP(1) THEN 10
0006  PRINT "FILE ";F$;" DOES NOT EXIST."
0007  GOTO 9999
0010  FILE #1,6;" "
.
.
.
.
.
0100  IF REF(6) #46 THEN 200
0101  PRINT "NO WRITE ACCESS TO FILE ";F$
0102  RESUME 9999
0200  ON ERROR THEN 0
9999  END
```

Line 4 attempts to link the file in Mode 3. If the file exists, and is shared with the present user, even on a READ ONLY basis, the TYP(1) in Line 5 will yield a positive number, and the attempt will be made to link the file in Mode 6. If the file has been shared READ ONLY, then Line 10 will generate an error code 46 — BUSY/PROTECTED FILE which will be picked up by the error routine. If an error other than 46 is encountered, the user will be told the line number in which the error occurred, and then the error routine and the program will be exited.

Clearly, to facilitate the cooperative and simultaneous use of files, it is desirable to relinquish WRITE links as soon as they are not needed. Mode 5 (READ ONLY) enables rapid-fire switching between READ/WRITE and READ ONLY links to a file. Modes 5 and 6 execute more rapidly than do Modes 1 through 4 because only the file number is referenced, and not the file name.

The following program reads pairs of records from a file, and based on the data in record A, takes the appropriate action to update record B. Specifically, the file MYFILE has 10 records. Each odd-numbered record contains either a string "I" (for "increment") or a string "D" (for "decrement"). Each even-numbered record contains an integer between 1 and 10. The integer in the even-numbered record is either increased by 1 or decreased by 1, depending on its present value, and on the value of the string in the odd-numbered record which precedes it.

```
LIS
PROG1

ØØ1Ø   FILE #1,1;"MYFILE"
ØØ2Ø   ON  END #1 THEN 21Ø
ØØ3Ø   X=1
ØØ4Ø   Y=2
ØØ5Ø   FILE #1,6;""
ØØ6Ø   READ #1,X;P$
ØØ7Ø   READ #1,Y;N
ØØ8Ø   IF (N<1Ø) AND (N>1) THEN  IF P$="D" THEN 16Ø
ØØ9Ø   IF (N<1Ø) AND (N>1) THEN  IF P$="I" THEN 14Ø
Ø1ØØ   IF N<2 THEN 13Ø
Ø11Ø   PRINT #1,X;"D"
Ø12Ø   GOTO 16Ø
Ø13Ø   PRINT #1,X;"I"
Ø14Ø   PRINT #1,Y;N+1
Ø15Ø   GOTO 17Ø
Ø16Ø   PRINT #1,Y;N-1
Ø17Ø   FILE #1,5;""
Ø18Ø   X=X+2
Ø19Ø   Y=Y+2
Ø2ØØ   GOTO 5Ø
Ø21Ø   PRINT "UPDATE COMPLETE..."
Ø22Ø   PRINT
Ø23Ø    FOR I=1 TO 1Ø STEP 2
Ø24Ø    READ #1,I;A$
Ø25Ø    READ #1,(I+1);A
Ø26Ø    PRINT A$;A;"  ";
Ø27Ø    NEXT I
Ø28Ø   END

RUN
PROG1

UPDATE COMPLETE...

I 5    D 6    D 9    I 2    D 4
DONE
RUN
PROG1

UPDATE COMPLETE...

I 6    D 5    D 8    I 3    D 3
DONE
RUN
PROG1

UPDATE COMPLETE...

I 7    D 4    D 7    I 4    D 2
DONE
RUN
PROG1

UPDATE COMPLETE...

I 8    D 3    D 6    I 5    D 1
DONE
```


BEGIN
LINK FILE TO READ ONLY (LINE 10)
SET END-OF-FILE FLAG (LINE 20)
RE-LINK FILE TO READ/WRITE (LINE 50)
READ NEXT RECORD PAIR P$ - FIRST RECORD N = SECOND RECORD (LINES 60 AND 70)
N <10 AND N >1 AND P$ = "D" (LINE 80)
YES
A
NO
N <10 AND N > 1 AND P$ = "I" (LINE 90)
YES
B
NO
N <2 (LINE 100)
NO
YES
SET P$ = "D" WRITE BACK TO FILE (LINE 110)
SET P$ = "I" WRITE BACK TO FILE (LINE 130)
A
B
DECREMENT N BY 1 AND WRITE BACK TO FILE (LINE 160)
INCREMENT N BY 1 AND WRITE BACK TO FILE (LINE 140)
RE-LINK FILE TO READ ONLY (LINE 170)
OUTPUT "UPDATE COMPLETE" TO TERMINAL (LINE 210)
READ NEW VALUES FROM FILE AND OUTPUT TO TERMINAL (LINES 230 TO 270)
END

Chapter 7

COMMAND FILES

A command file is a series of BASIC-X commands and/or program statements and/or program input which is stored in an ordinary BASIC-X file. Command files are said to be "executed" instead of being "read from" or "written to". This is because the commands and program statements in the file are executable directly from the file, as though they were being read from paper tape. The COMmand command is used to execute a command file and the syntax is,

COM-*filename*

FEATURES

1. The command file is composed of strings. Each string in the file is a separate command, statement or program input line. Any numeric values in the file are ignored by the COMmand command.
2. The strings are stored in a command file in the normal way, via a program.
3. Command files are not linked during their execution, and could be closed or written to at that time. One of two error messages would be generated in this event:

   COMMAND FILE MISSING

   and

   COMMAND FILE MODIFIED

   mean that during execution of a command file the system referenced the file for the next input line and discovered that the file had been accessed from another port.
4. Command files residing in libraries other than the current user's own may be accessed if they are shared READ ONLY or READ/WRITE, as follows:

   COM-$$$*file*

   executes a command file in the System public library

   COM-$$*file*

   executes a command file in the Master public library

   COM-$*file*

   executes a command file in the Group public library

   COM-#*account,file*

   executes a command file in another user's library.

Command files are executed serially beginning with the first record in the file. Execution of a command file is stopped when the system encounters:

A. the end of the file
B. a file mark in the file
C. an abort (stopping a program with two BREAKs)
D. an Execute Immediate condition (one BREAK or executing a STOP statement)
E. a syntax error
F. a COM command with a *filename* parameter (a COM command without a *filename* parameter, encountered during execution of a command file, is ignored)

If execution of a command file is stopped before the end of the file is reached, execution of the file may be resumed by typing COM without the *filename.*

Command files cannot be "nested". That is, if a COMmand command with a *filename* parameter occurs in a command file, the system stops executing the current command file and begins executing the next command file.

Execution of a command file is stopped if the system encounters a syntax error, and the system prints an error message, but not the number of the line in which the error occurred, unless the program entry is preceded by the TAPe command. For example, if the following strings are stored in a file named EXEC:

```
TAP
10    PRINT "EXAMPLE
20    END
KEY
```

then the command COM-EXEC will produce the following result:

```
NO CLOSING QUOTE IN LINE 10
LAST INPUT IGNORED, RETYPE IT
```

then the system will read and attempt to execute the next string in the command file. This would be inconvenient if, for example, the next string in the command file were RUN. Without the TAPe command, the output from incorrectly entering Line 10 would be:

```
NO CLOSING QUOTE
```

and the execution of the command file will stop. In either case the improperly typed line will be ignored by the system and not stored as part of the program.

Errors encountered during execution of a program in a command file will invariably cause execution of the command file to stop and the appropriate error message, including line number, to be printed, whether or not the TAPe command was used during program entry. The only exception to this is in

the case of WARNING ONLY errors which do not cause the command file to stop. For example, if the following strings are stored in a file named EXEC:

```
TAP
1Ø    PRINT B
2Ø    END
KEY
RUN
```

then the command COM-EXEC will produce the following result:

```
UNDEFINED VALUE ACCESSED IN LINE 1Ø
STOPPED AT 2Ø
XI?
```

At this point, execution of the command file is stopped, and the program error has caused the system to go to Execute Immediate mode. All the options normally available in Execute Immediate are open to the user. He may, if he wishes, modify the program, RUN it, and then type

```
COM
```

to continue execution of the command file.

The following is typical of a program that might be used to write strings to a command file:

```
0010  DIM A$[254]
0020  FILE #1,2;"EXEC"
0030  READ #1,1
0040  ON  END #1 THEN 200
0050  INPUT "COMMAND: ",A$
0060  IF A$="" THEN 9999
0070  PRINT #1;A$
0080  GOTO 50
0200  PRINT "END OF FILE"
9999  END
```

This program permits the user to specify both the file name and the file commands interactively. The program could just as well specify these items internally. For example,

```
0010  FILE #1,2;"EXEC"
0020  PRINT #1;"SCR"
0030  PRINT #1;"TAP"
0040  PRINT #1;"10 DIM B(5),C(5)"
0050  PRINT #1;"20 MAT READ B"
0060  PRINT #1;"30 DATA 20,63,407,811,6"
0070  PRINT #1;"35 RESTORE"
0080  PRINT #1;"40 MAT READ C"
0090  PRINT #1;"50 MAT C=B+C"
0100  PRINT #1;"60 MAT PRINT C"
0110  PRINT #1;"70 END"
0120  PRINT #1;"SAV-MATMULT"
0130  PRINT #1;"RUN"
9999  END
```

It is good practice to precede entry of program statements with the SCRatch command as in Line 20 above, so as to avoid any possibility of overlaying a program currently in the work area with a program defined within the command file. Executing the file named EXEC will produce the following result:

```
COM-EXEC

 40
 126
 814
 1622
 12
 DONE
```

Appendix A

COMMANDS

Commands fall into three categories:

1. System Commands
2. Edit Commands
3. Library Commands

In general, the format for issuing any command is,

COMmand-*parameter,parameter*

Only the first three letters of any command are necessary for the system to recognize the word. For this reason, the syntax illustrations for commands are shown with the first three letters capitalized and the rest in lower case. Actually, the system recognizes upper or lower case, and the entire command word may be typed, if desired.

Some commands may be used without explicit parameters. In these cases the parameters have default values, and only the command word itself need be typed if the user finds the default values acceptable. Where explicit parameters are used, the command word must be followed by a hyphen (-) and then by the parameter(s). Where multiple parameters are specified to a single command, the parameter values must be separated by commas.

Execution of a command is followed by a system-generated X-ON (tape reader ON) character. This means that a series of commands could be punched on paper tape and then executed without intervention from the user. For example, the following series might be executed from paper tape:

1. HEL-A123
2. GET-MYPROG
3. RUN-100
4. 14327
5. 85
6. 607
7. BYE

Lines 4 through 6 represent data to be typed in response to INPUT statements in the program named MYPROG. The remaining lines, 1 – 3 and 7 are commands. Because the system generates X-ON characters after it executes each command, each line from the tape is read automatically. For further information regarding the use of paper tape, see Appendix D.

SYSTEM COMMANDS

1. BYE

   logs the user off the system, and turns off any active command files for that port. The BYE command has no parameter.

2. CAR-*N* (CARriage-*N*)

   sets the logical line width in number of characters per line. N must be an integer in the range 20 to 255, or it must be 0. The command CAR-0 sets the carriage width to "unlimited". This feature is used for plotting on terminals, for CRT cursor control, for backspacing and for various other special applications. The default setting of the CAR command is 72.

3. CRD-*N*

   sets the Carriage Return Delay to N character times. For example, if the port is operating at 10 characters per second (10 CPS or 110 baud) then CRD-25 will set the carriage return delay to two and one half seconds. The CRD command must always have one and only one explicit parameter. N must be in the range 0 to 100, and the default value is 0.

4. DAT (DATe)

   gives today's date and the time of day, e.g.,

   ```
   DAT
   TUE 11-MAY-76 03:45PM
   ```

5. ECHO-OFF

   and

   ECHO-ON

   All terminals are assumed by the system to be operating in ECHO-PLEX (sometimes also called FULL DUPLEX) mode. If there is a "HALF/FULL" switch on the terminal it should be set to "FULL". If there is no switch and the terminal operates only in HALF DUPLEX mode, then the user should adjust the computer's transmission mode by typing,

   ```
   ECHO-OFF
   ```

   Echo-plex transmission is restored by typing

   ```
   ECHO-ON
   ```

   OFF and ON are the only two parameters available to the ECHO command.

6. FFD-*N*

   sets the Form Feed Delay where N = character times as for the CRD command. The FFD command must always have one and only one explicit parameter. N must be in the range 0 to 100, and the default value is 0.

7. HEL-*account,password* (HELlo-*account,password*)

logs the user onto the system. Both parameters must be explicit and must have a specified form:

a. Account is the user's account number. It must be a one letter, three digit combination, e.g.,

X123

b. Password is the "lock" on the account number which helps prevent unauthorized use of the account. For further information, see the section on Log ON/OFF procedures.

8. LFD-*N*

sets the Linefeed Delay where N = character times as for the CRD command. The LFD command must always have one and only one explicit parameter. N must be in the range 0 to 255, and the default value is 0.

9. RAT-*N*! ! (RATe-*N*! !)

The rate of character transmission is set port by port at the computer. Once logged on, the user may adjust the baud rate via the RATe command.

N is the desired baud rate. The exclamation points are used only if the desired baud rate or bits/word is non-standard. The standard rates are, 110 baud/11-bit words and 150, 220, 300, 440, 600, 880, 1200 and 2400, all with 10-bit words. If the desired rate is 880 baud with 10-bit words, then the command would be typed,

RAT-880

and if the desired rate is 880 with 11-bit words then the command would be typed,

RAT-880!!

Any time a non-standard baud rate is chosen, there must be at least one exclamation point to tell the computer explicitly which bit pattern is desired. The user is cautioned that if the rate is set on a hardwired terminal to some rate that the terminal can't use, the terminal must be evicted by the System Manager in order to restore the default rate of 110 baud/11-bit characters.

10. ROS (ROSter)

shows which port(s) the inquiring account number is using, the other ports currently in use (++++), and the other inactive, available ports (. . . .). In a sixteen port system, for example,

```
ROS
++++  ++++  ....  A123  ++++  ....  ....  ....
....  ....  ....  ++++  ++++  ++++  ....  A123
```

shows that account A123 is logged on to ports 3 and 15, other users are logged on to ports 0, 1, 4, 11, 12 and 13, and the remaining ports are unoccupied.

11. TIM (TIMe)

gives in hours to the nearest hundredth,

connect time since last log-on

on-port time since last system re-set

off-port time since last system re-set

total on-port plus off-port time

On-port time is time used by an account on the port dedicated to that account, if one exists. Off-port time is time used by an account on any port other than a dedicated port. If there is no dedicated port for a given account, then all time used by that account is considered to be off-port time. For example,

```
TIM
TIME IN HOURS:

CONNECT  =  000.04
ON-PORT  =  000.00
OFF-PORT =  005.23
TOTAL    =  005.23
```

EDIT COMMANDS

The following commands modify or refer to the BASIC program currently in the user's work area.

1. DEL-*M*,*N* (DELete-*M*,*N*)

erases lines M through N of the current program. The N parameter is optional and if omitted, then lines M through the last line in the program are erased.

2. LEN (LENgth)

prints the length, in words, of the current program. SOURCE LENGTH reflects the amount of disk space required to SAVe the program. If the program has been RUN, then the COMPILED LENGTH is also printed, and reflects the amount of memory the program requires while running.

Some guidelines for calculating space requirements are:

1. there are about 10,112 words of memory available to the user

2. 2 words are required for each defined variable and each defined array element
3. about 11 to 14 words are used per average BASIC program statement
4. each character in a string requires 1/2 word, and each string requires 1 + INT [(N+1)/2] words
5. each open file requires a 134-word buffer (linking a file with reference number 6, for example, creates one 134-word buffer for each of files 1, 2, 3, 4 and 5 if they are not already linked, in addition to the 134-word buffer for file 6.)
6. REM statements require 3 words plus 1 word for every two characters in the comment

The LENgth command has no parameter.

3. KEY

removes the user from the TAPe mode. The KEY command has no parameter.

4. LIS-*N* (LISt-*N*)

causes the system to list the current program. The N parameter is optional and if specified, the program is listed starting with statement number N. If the parameter is not specified, listing starts with the first statement. In either case the program is listed to its last statement.

5. NAM-*PROGRAM* (NAMe-*PROGRAM*)

gives the current program the name specified by the word following the command, in this case, *PROGRAM*. The program name may be up to 10 characters excepting blanks, quotation marks, commas and non-printing characters. The first character of the program name may not be $ or #.

6. PUN-*N* (PUNch-*N*)

causes the current program to be punched on paper tape. The PUNch command works the same way as the LISt command insofar as the parameter specification is concerned. Output sequence from the command is,

linefeed
program name
tape on (R^C)
leader
program
X-OFF
trailer
tape off (T^C)

7. REN-*new, increment, start, stop* (RENumber-*new,increment,start,stop*)

renumbers all or some statements in the current program. All four parameters are optional and if omitted, the default values are,

REN-10,10,1,9999

If the command parameters are specified in such a way as to threaten the existing order of program statements, the system responds,

SEQUENCE NUMBER OVERLAP

Any line number references in the program such as in GOTO statements are changed to match the renumbered statements, if appropriate.

8. RUN-*N*

The current program is executed. The N parameter is optional and, if omitted, the program is executed beginning with the first statement. If the N parameter is specified then the program is executed starting with that statement number or the next higher existing statement number.

9. SCR (SCRatch)

erases the current program and program name from the user's work area. The SCRatch command has no parameter.

10. TAP (TAPe)

prepares the computer to receive a program input from paper tape. The system remains in tape mode until the user explicitly changes the mode, usually by the KEY command.

Syntax error messages are suppressed until program entry is complete, but statements containing errors are automatically deleted from the program. If the system found errors during input, then when a system command is entered the error messages will be printed out. If no error messages are pending then the system command will be executed. The KEY command is used for this purpose. The TAPe command has no parameter.

11. XPU (XPUnch)

works exactly the same as the PUNch command except that an X-OFF is punched immediately following each program statement, just before the carriage return and linefeed.

LIBRARY COMMANDS

1. ABO-*program* (ABOrtable-*program*)

restores abortable status to a formerly non-abortable program. The ABOrtable command is effective when issued by the same account which rendered the program non-abortable, or by a higher-level account.

2. APP-*programname* (APPend-*programname)*

retrieves and appends the referenced program to the current program in the user's work area. The first statement number of the referenced program must be higher than the last statement number of the current program, otherwise the system will respond:

SEQUENCE NUMBER OVERLAP

and the command will be ignored. $ symbols preceding the program name are treated as they are in the GET command.

To reference a program which resides in a foreign library and which has been shared with the current user, the syntax is,

APP-#*account,program*

The user is cautioned that if the referenced program is RUN ONLY then the (now combined) current and appended programs will be RUN ONLY, and not available to LISt, PUNch, XPUnch, DELete, or any editing command.

3. CAT-*M,N* (CATalog-*M,N*)

prints out detailed information about the user's own library. Both parameters are optional and refer to the serial number of the catalog entries, the newest entries coming highest in the list. If the N parameter is omitted then the catalog listing begins with the serial number specified by M and continues to the earliest entry in the catalog. If the N parameter is specified then the serial numbers of the entries in the range M through N are printed. If both parameters are omitted then the entire catalog is printed. For example,

```
CAT
 S/N     NAME        CODE   LEN.   SECT. SAVED    ACCESS    USE

00089   EVERYONE     7N7   00003   0001  191/75   191/75   00000
00088   PER@005       P4   00003   0001  191/75   191/75   00000
00087   PRO@005      4P    00003   0001  191/75   191/75   00000
00086   PER@002       P2   00003   0001  191/75   191/75   00000
00085   PRO@002      2P    00003   0001  191/75   191/75   00000
00084   PERMASTER     P1   00003   0001  191/75   191/75   00000
00083   PROMASTER    1P    00003   0001  191/75   191/75   00000
00082   PK/MODE       P    00320   0003  188/75   188/75   00000
00080   EXAMPLE       P    00249   0003  182/75   182/75   00000
00079   TESTFILE      F    00010   0011  182/75   182/75   00001
00071   TEST          F    00010   0011  182/75   182/75   00004
00070   TESTPROG      P    00036   0001  181/75   181/75   00000
00064   PRINTUSING    P    00205   0002  178/75   178/75   00001
00058   PS8-1         P    00046   0001  171/75   171/75   00000
00057   MAT           P    00062   0001  167/75   167/75   00003
00056   TESTREC4      P    00060   0001  162/75   162/75   00000
00054   DUMMY4        F    00001   0002  162/75   162/75   00008
00053   TESTREC3      P    00163   0002  162/75   162/75   00001
00052   TESTREC2      P    00112   0001  162/75   162/75   00004
```

```
00051  DUMMY3        F    00001  0002  162/75  162/75  00020
00050  DUMMY2        F    00005  0006  162/75  162/75  00006
00049  TESTREC       P    00142  0002  162/75  162/75  00003
00048  DUMMY1        F    00005  0006  162/75  188/75  00034
00047  PROG2         P    00030  0001  156/75  156/75  00001
00046  PROG1         P    00074  0001  156/75  156/75  00002
00042  MATMULT1      P    00165  0002  155/75  155/75  00000
00041  MATMULT2      P    00213  0002  155/75  155/75  00003
00038  RANDOM        P    00033  0001  153/75  153/75  00001
00029  MATS          P    00110  0001  143/75  143/75  00001
00028  STRINGMATH    P    00044  0001  141/75  143/75  00002
00024  FX2           P    00279  0003  114/75  136/75  00007
00023  FX1           P    00386  0004  114/75  136/75  00002
00010  DISCOUNT      P    00226  0002  099/75  099/75  00000

TOTAL STORAGE = 00080 SECTORS

CAT-89,86

 S/N    NAME        CODE  LEN.   SECT. SAVED   ACCESS   USE

00089  EVERYONE     7N7  00003  0001  191/75  191/75  00000
00088  PER@005       P4  00003  0001  191/75  191/75  00000
00087  PRO@005      4P   00003  0001  191/75  191/75  00000
00086  PER@002       P2  00003  0001  191/75  191/75  00000

CAT-84

 S/N    NAME        CODE  LEN.   SECT. SAVED   ACCESS   USE

00084  PERMASTER     P1  00003  0001  191/75  191/75  00000
00083  PROMASTER    1P   00003  0001  191/75  191/75  00000
00082  PK/MODE       P   00320  0003  188/75  188/75  00000
0

ABORT
```

S/N is the sequence in which the catalog entry was made
NAME is the name of the file or program entered in the catalog
CODE indicates whether the catalog entry is a file or a program

The numbers to the left and right of the code letters give the type and level of protection

1 is Master Account protection

2 is System Manager protection

3 is Master Account and System Manager protection

4 is factory protection

5 is factory and Master Account protection

6 is factory and System Manager protection

7 is factory, System Manager and Master Account protection

For programs, the code letter N means that the program is non-abortable, and the code letter P means that the program is abortable. A number to the left of the letter means that the program cannot be LISted, and a number to the right of the

letter means that the program cannot be KILled. For files, the code letter F means that the file is OPEn, and a code letter C means that the file is being CLOsed. A number to the left of the letter means that the file is READ ONLY, and a number to the right of the letter means that the file cannot be CLOsed.

LEN. is the number of words of storage required to save a program or the number of records available in a file

SECT. is the number of disk sectors (records) used to save the program or file

SAVED is the date on which the program or file was last saved

ACCESS is the date on which the program or file was last accessed

USE is the number of accesses since the program or file was saved or opened, respectively

4. CLO-*filename* (CLOse-*filename*)

deletes the referenced file from the user's library and overwrites it with End-Of-File marks.

5. GET-*programname*

retrieves the reference program into the user's work area. To GET a program which resides in a foreign library and which has been shared with the current user, the syntax is,

GET-#*account,programname*

When the programname parameter is preceded by one $ symbol, the Group Library is searched. Two $ symbols cause the Master Account Library to be searched for the program. When three $ symbols precede the program name, the System Public Library is searched. If no $ symbol precedes the program name then the user's own private library is searched.

6. HID-*name* (HIDe-*name*)

veils the named file or program from the INDEX command but not from the CATalog command

7. IND (INDex)

gives a summary listing of the contents of the user's library. File names are preceded by an asterisk.

8. KIL -*programname* (KILl-*programname*)

erases the referenced program from the user's library.

9. NON-*programname*

renders the referenced program non-abortable. If a master account desires to make a program in another library non-abortable then the command must be typed,

NON-#*account,programname*

10. OPE-*filename, size* (OPEn-*filename,size*)

creates a file with the chosen name and the specified number of records. Filename may be

up to ten printing characters excepting commas, quotation marks, embedded blanks, lower case letters and leading # or $ symbols. Size is the maximum number of records that may be filled with data for that file. It must be in the range 1 to 9502, and is functionally dependent on the available disk space and/or the user's storage limit, whichever is smaller.

11. RED-*oldname,newname* (REDesignate-*oldname,newname*)

changes the name of the specified file or program from *oldname* to *newname.* The Use Counter is set to Ø and the date Accessed is set to the date that the RED command is executed. The entry's Saved date and Serial Number are unchanged.

12. SAV-*programname* (SAVe-*programname*)

adds the current program to the user's library. The programname parameter is unnecessary if the current program already has an acceptable name. (See the NAMe command.) While it is legal to name and save the current program in the same execution of the SAVe command, a name used in a SAVe command has no effect on the name of the copy of the program still in the user's work area. If there already exists a program with the same name in the user's library the system will respond:

DUPLICATE ENTRY

if the user wishes to replace the existing program in his library with the current program the command sequence is,

KIL-*programname*

SAV-*programname*

13. SEE-*name*

unveils the named file or program to the INDex command.

14a. SHA-*filename,account*,R (SHAre-*filename,account*,R)

shares the referenced file with the named account. The R parameter is optional, and grants READ ONLY access to the file. The file owner still retains READ/WRITE access to the file in this case. If the file is shared with account @001 then it is in effect, shared with all accounts.

14b. SHA-*program,account* (SHAre-*program,account*)

shares the named program with the named account.

15. UNS-*name,account* (UNShare-*name,account*)

revokes all shared access privileges to the name program or file for the named account. Although the form UNS-filename,@001 does in effect revoke all shared privileges to all accounts, it is only effective if the original SHAre command used the @001 account number. To revoke the entire shared list for a file or program:

UNS-*name*

Appendix B

ASCII CHARACTER SET

DEC	OCT	NAME	TTY KEYS	DEC	OCT	NAME	TTY KEYS	DEC	OCT	NAME	TTY KEYS	DEC	OCT	NAME	TTY KEYS
0	000	NUL	P(cs)	32	040	spa.		64	100	@	p(s)*	96	140		*
1	001	SOH	A(c)	33	041	!	1(s)	65	101	A		97	141	a	
2	002	STX	B(c)	34	042	"	2(s)	66	102	B		98	142	b	
3	003	ETX	C(c)	35	043	#	3(s)	67	103	C		99	143	c	
4	004	EOT	D(c)	36	044	$	4(s)	68	104	D		100	144	d	
5	005	ENQ	E(c)	37	045	%	5(s)	69	105	E		101	145	e	
6	006	ACK	F(c)	38	046	&	6(s)	70	106	F		102	146	f	
7	007	BEL	G(c)	39	047	'	7(s)	71	107	G		103	147	g	
8	010	BS	H(c)	40	050	(	8(s)	72	110	H		104	150	h	
9	011	HT	I(c)	41	051	)	9(s)	73	111	I		105	151	i	
10	012	LF	J(c)	42	052	*	:(s)	74	112	J		106	152	j	
11	013	VT	K(c)	43	053	+	;(s)	75	113	K		107	153	k	
12	014	FF	L(c)	44	054	,		76	114	L		108	154	l	
13	015	CR	M(c)	45	055	-		77	115	M		109	155	m	
14	016	SO	N(c)	46	056	.		78	116	N		110	156	n	
15	017	SI	O(c)	47	057	/		79	117	O		111	157	o	
16	020	DLE	P(c)	48	060	0		80	120	P		112	160	p	
17	021	DC1	Q(c)	49	061	1		81	121	Q		113	161	q	
18	022	DC2	R(c)	50	062	2		82	122	R		114	162	r	
19	023	DC3	S(c)	51	063	3		83	123	S		115	163	s	
20	024	DC4	T(c)	52	064	4		84	124	T		116	164	t	
21	025	NAK	U(c)	53	065	5		85	125	U		117	165	u	
22	026	SYN	V(c)	54	066	6		86	126	V		118	166	v	
23	027	ETB	W(c)	55	067	7		87	127	W		119	167	w	
24	030	CAN	X(c)	56	070	8		88	130	X		120	170	x	
25	031	EM	Y(c)	57	071	9		89	131	Y		121	171	y	
26	032	SUB	Z(c)	58	072	:		90	132	Z		122	172	z	
27	033	ESC	K(cs)	59	073	;		91	133	[	K(s)*	123	173	{	*
28	034	FS	L(cs)	60	074	<	,(s)	92	134	\	L(s)*	124	174	|	*
29	035	GS	M(cs)	61	075	=	-(s)	93	135	]	M(s)*	125	175	}	*
30	036	RS	N(cs)	62	076	>	.(s)	94	136	↑	N(s)*	126	176	~	*
31	037	US	O(cs)	63	077	?	/(s)	95	137	←	O(s)*	127	177	DEL	*

*The location of these keys may vary on some ASCII keyboards.

LEGEND

DEC	NAME	DESCRIPTION	DEC	NAME	DESCRIPTION	DEC	NAME	DESCRIPTION
0	NUL	Null	11	VT	Vertical Tab	22	SYN	Synchronous Idle
1	SOH	Start of Heading	12	FF	Form Feed	23	ETB	End of Transmission Blk.
2	STX	Start of Text	13	CR	Carriage Return	24	CAN	Cancel
3	ETX	End of Text	14	SO	Shift Out	25	EM	End of Medium
4	EOT	End of Transmission	15	SI	Shift In	26	SUB	Substitute
5	ENQ	Enquiry	16	DLE	Data Link Escape	27	ESC	Escape
6	ACK	Acknowledgment	17	DC1	Device Control 1	28	FS	File Separator
7	BEL	Bell	18	DC2	Device Control 2	29	GS	Group Separator
8	BS	Backspace	19	DC3	Device Control 3	30	RS	Record Separator
9	HT	Horizontal Tab	20	DC4	Device Control 4	31	US	Unit Separator
10	LF	Line Feed	21	NAK	Neg. Acknowledge	127	DEL	Delete (Rubout)

NOTE: (s) Means Hold Shift Key Down
(c) Means Hold Control Key Down
(cs) Means Hold Both Keys Down

Appendix C

ERROR MESSAGES

The following is a list of program error messages that a user might encounter during program entry or program execution. The messages are logically grouped by program statement type.

SYNTAX ERRORS

1. OUT OF STORAGE
2. ILLEGAL OR MISSING INTEGER
3. EXTRANEOUS LIST DELIMITER — usually extra punctuation in a file operation
4. MISSING ASSIGNMENT OPERATOR
5. CHARACTERS AFTER STATEMENT END — usually extraneous characters in a list of variables or data
6. MISSING OR ILLEGAL SUBSCRIPT
7. MISSING OR BAD LIST DELIMITER
8. MISSING OR BAD FUNCTION NAME
9. MISSING OR BAD SIMPLE VARIABLE
10. MISSING OR ILLEGAL 'OF'
11. MISSING OR ILLEGAL 'THEN'
12. MISSING OR ILLEGAL 'TO'
13. MISSING OR ILLEGAL 'STEP'
14. MISSING OR ILLEGAL DATA ITEM
15. ILLEGAL EXPONENT
16. SIGN WITHOUT NUMBER
17. MISSING RELATIONAL OPERATOR
18. ILLEGAL READ VARIABLE
19. ILLEGAL SYMBOL FOLLOWS 'MAT' — matrix operation error
20. MATRIX CANNOT BE ON BOTH SIDES — matrix operation error
21. NO '*' AFTER RIGHT PARENTHESIS — matrix operation error
22. NO LEGAL BINARY OPERATOR FOUND — matrix operation error
23. MISSING LEFT PARENTHESIS
24. MISSING RIGHT PARENTHESIS
25. PARAMETER NOT STRING VARIABLE
26. UNDECIPHERABLE OPERAND
27. MISSING OR BAD ARRAY VARIABLE
28. STRING VARIABLE NOT LEGAL HERE

29. MISSING OR BAD STRING OPERAND
30. NO CLOSING QUOTE
31. 254 CHARACTERS MAX FOR STRING
32. STATEMENT HAS EXCESSIVE LENGTH
33. ILLEGAL STATEMENT TYPE
34. OVER/UNDERFLOW(S)
35. UNDEFINED SYMBOL
36. MISSING OR BAD FILE REFERENCE

RUN ERRORS

37. UNDEFINED STATEMENT REFERENCE
38. NEXT WITHOUT MATCHING FOR
39. SAME FOR-VARIABLE NESTED — two or more 'NEXT' statements attempt to reference the same 'FOR' variable
40. FUNCTION DEFINED TWICE
41. VARIABLE DIMENSIONED TWICE
42. LAST STATEMENT NOT 'END'
43. UNMATCHED FOR — 'FOR' statement has no matching 'NEXT' statement
44. UNDEFINED FUNCTION
45. ARRAY TOO LARGE
46. ARRAY OF UNKNOWN DIMENSIONS
47. OUT OF STORAGE
48. DIMENSIONS NOT COMPATIBLE
49. CHARACTERS AFTER COMMAND END — usually a parameter declared where none is allowed
50. INVALID FILE MODE — attempt to link file with meaningless access mode code
51. NO SUCH FILE
52. GOSUBS NESTED TEN DEEP — nine is the maximum
53. RETURN WITH NO PRIOR GOSUB
54. SUBSCRIPT OUT OF BOUNDS
55. NEGATIVE STRING LENGTH — expression attempts to define a string with fewer than zero characters
56. NON-CONTIGUOUS STRING CREATED — expression attempts to define a string with voids between some characters
57. STRING OVERFLOW
58. OUT OF DATA

59. DATA OF WRONG TYPE
60. UNDEFINED VALUE ACCESSED
61. MATRIX NOT SQUARE — certain matrix operations require square matrices
62. REDIMENSIONED ARRAY TOO LARGE
63. NEARLY SINGULAR MATRIX — expression results in attempt to divide by zero
64. LOG OF NEGATIVE ARGUMENT
65. ARGUMENT OUT OF BOUNDS
66. ZERO TO ZERO POWER
67. NEGATIVE NUMBER TO REAL POWER
68. ARGUMENT OF SIN OR TAN TOO BIG
69. BAD FILE NUMBER
70. LAST INPUT IGNORED, RETYPE IT — usually due to static in communications lines
71. STATEMENT NOT IMAGE — output formatting error
72. NON-EXISTENT FILE REFERENCED
73. BUSY/PROTECTED FILE
74. NO SUCH PROGRAM
75. CHAINED PROGRAM TOO LARGE — can be caused by defining several very large arrays in the program, because buffer space is not set aside until run time
76. WRITE TRIED TO READ-ONLY FILE
77. END-OF-FILE/END-OF-RECORD
78. FILE ABNORMAL/UNAVAILABLE
79. DONE
80. MISSING FORMAT SPECIFICATION
81. ILLEGAL OR MISSING DELIMITER
82. NO CLOSING QUOTE
83. BAD CHARACTER AFTER REPLICATOR — output formatting error
84. REPLICATOR TOO LARGE — output formatting error
85. REPLICATOR ZERO — output formatting error
86. MULTIPLE DECIMAL POINTS
87. BAD FLOATING POINT SPEC. — output formatting error
88. ILLEGAL CHARACTER IN FORMAT — output formatting error
89. ILLEGAL FORMAT FOR STRING — output formatting error
90. MISSING RIGHT PARENTHESIS
91. MISSING REPLICATOR — output formatting error
92. TOO MANY PARENTHESIS LEVELS

93. MISSING LEFT PARENTHESIS
94. ILLEGAL FORMAT FOR NUMBER — output formatting error
95. BAD FORMAT FOR STRING NUMBER — misuse of string arithmetic capability

WARNING MESSAGES

96. BAD INPUT, RETYPE FROM ITEM
97. LOG OF ZERO — WARNING ONLY
98. ZERO TO NEGATIVE POWER-WARNING
99. DIVIDE BY ZERO — WARNING ONLY
100. EXP OVERFLOW — WARNING ONLY
101. OVERLFOW — WARNING ONLY
102. UNDERFLOW — WARNING ONLY
103. EXTRA INPUT — WARNING ONLY
104. BAD STRING — RETYPE FROM ITEM

The following error messages apply only if the issued command is in some way faulty. Otherwise the command is executed as it is specified.

GET

1. INVALID NAME
2. ACCOUNT NOT ON SYSTEM
3. NO SUCH PROGRAM
4. NOT A BASIC PROGRAM
5. ILLEGAL ACCESS
6. PROGRAM TOO LARGE — computer has old, 8800 word swap area

NAMe

1. ONLY 10 CHARACTERS ACCEPTED
2. $ OR # ILLEGAL AS FIRST CHARACTER

SAVe

1. $ OR # ILLEGAL AS FIRST CHARACTER
2. DISK FULL
3. EXCEEDS ALLOCATED SPACE ON THIS DISK

4. DIRECTORY FULL
5. DUPLICATE ENTRY — NO SAVE
6. NAME TOO LONG
7. NO PROGRAM NAME

RUN

1. NO PROGRAM
2. CHARACTER AFTER COMMAND END

APPend

1. INVALID NAME
2. SEQUENCE NUMBER OVERLAP
3. ACCOUNT NO ON SYSTEM
4. NO SUCH PROGRAM
5. NOT A BASIC PROGRAM
6. ILLEGAL ACCESS
7. PROGRAM TOO LARGE

DELete

1. NOTHING DELETED — no program, or the line number doesn't exist in the current program

KILl

1. INVALID NAME
2. NO SUCH PROGRAM
3. NOT A PROGRAM
4. NON-KILL PROGRAM

RENumber

1. SEQUENCE NUMBER OVERLAP
2. NO PROGRAM — there is no program currently in the user's work area

LISt

1. RUN ONLY
2. CHARACTER AFTER COMMAND END

PUNch

1. RUN ONLY
2. CHARACTER AFTER COMMAND END

XPUnch

1. RUN ONLY
2. CHARACTER AFTER COMMAND END

ABOrtable

1. INVALID NAME
2. ACCOUNT NOT ON SYSTEM
3. NO SUCH PROGRAM
4. NOT A BASIC PROGRAM
5. ILLEGAL ACCESS

NON (Non-abortable program)

1. INVALID NAME
2. ACCOUNT NOT ON SYSTEM
3. NO SUCH PROGRAM
4. NOT A BASIC PROGRAM
5. ILLEGAL ACCESS

OPEn

1. DISK FULL
2. EXCEEDS ALLOCATED SPACE ON THIS DISK
3. DIRECTORY FULL — a limit can be set on the number of entries in a user's directory
4. DUPLICATE ENTRY — a file by that name is already open

SHAre

1. NO SUCH FILE
2. ENTRY IS NOT A BASIC FILE
3. PUBLIC FILE
4. DUPLICATE ENTRY

5. SHARED LIST FULL
6. ILLEGAL REQUEST

UNShare

1. NO SUCH FILE
2. ENTRY IS NOT A BASIC FILE
3. PUBLIC FILE
4. WAS NOT SHARED
5. WAS NOT PUBLIC
6. ILLEGAL REQUEST

CLOse

1. INVALID NAME
2. NO SUCH FILE
3. NOT A BASIC FILE
4. NON-CLOSE FILE
5. FILE IN USE
6. CLOSED

CATalog

1. ILLEGAL REQUEST

CARriage

1. 255 IS MAXIMUM
2. 40 IS MINIMUM

CRD (Carriage Return Delay)

1. MAX. DELAY = 100 CHARACTER TIMES

FFD (Form Feed Delay)

1. MAX. DELAY = 255 CHARACTER TIMES

LFD (Linefeed Delay)

1. MAX. DELAY = 100 CHARACTER TIMES

RATe

1. 2500 BAUD IS MAXIMUM RATE
2. 100 BAUD IS MINIMUM RATE
3. ALLOWED BAUD RATES ARE:

110	220	440	880	1760
150	300	600	1200	2400

GENERAL SYSTEM ERROR MESSAGES

ABORT — an executing program is terminated non-normally

PLEASE LOG IN — work attempted without prior user identification

EH? — any unrecognized command

RUN ONLY — attempted to add to or delete from a protected program

SWAP ERROR — internal computer error, advise System Manager

BAD FORMAT — usually too few or too many parameters specified

NOT IMPLEMENTED YET — for commands currently under development

Appendix D

PUNCHED PAPER TAPE

Among the uses for punched paper tape are: a) as a convenient and economical storage medium for infrequently used programs; b) paper tape containing program input data can be prepared and corrected off-line and then input to an executing program cleanly and quickly saving connect time that might otherwise be wasted on input errors and "ponder loops"; c) programs may be edited by punching them on paper tape and then reading them into an editing program line by line as character string data.

I The X-ON, X-OFF feature

Some terminals are equipped with automatic paper tape reader control. If present, this control allows the paper tape reader to be turned on and off under program control, and is activated by means of certain control characters on the terminal. (Control characters are formed by pressing the CONTROL key and while holding it down, striking the indicated letter key.) The control characters relevant to the paper tape reader are,

Q^C — activates the paper tape reader (commonly called X-ON)

S^C — de-activates the paper tape reader (commonly called X-OFF)

II Creating a punched paper tape

Paper tapes containing program listings may be created on-line by the PUNch or the XPUnch command.

A. PUNCH

The PUNch command is used when the program will be read back into the system all at once rather than one line at a time (as for data). It does not require the X-ON, X-OFF feature on the terminal. To use the PUNch command:

1. The terminal LOCAL/OFF/LINE switch is turned to LINE.
2. The user is logged on.
3. There is a program currently in the user's work area.
    a. Type PUN but do *not* strike the RETURN key.
    b. Turn the paper tape punch to the ON position.
    c. Strike the RETURN key.
    d. The paper tape is punched.
    e. Turn the paper tape punch to the OFF position.

The output to the paper tape will be,

tape leader
program statement
carriage return
linefeed
program statement
carriage return
linefeed
.
.
.
program statement
carriage return
linefeed
X-OFF (S^C)
tape trailer

B XPUnch

The XPUnch command works much the same way as the PUNch command except that an X-OFF character is inserted after each program statement, before the carriage return. This makes the program look like data to the computer, and is suitable for punching a program which will be edited by another, executing program, but is unsuitable for punching an executable program. A program tape prepared by the XPUnch command must be read to an editing program only on a terminal equipped with the X-ON, X-OFF feature. To XPUnch a program tape:

1. The terminal LOCAL/OFF/LINE switch is turned to LINE.
2. The user is logged on.
3. There is a program in the user's work area.
   a. Type XPU but do *not* strike the RETURN key.
   b. Turn paper tape punch to the ON position.
   c. Strike the RETURN key.
   d. The paper tape is punched.
   e. Turn the paper tape punch to the OFF position.

The output to the paper tape will be:

tape leader
program statement
X-OFF (S^C)
carriage return
linefeed
program statement
X-OFF (S^C)
carriage return
linefeed
.
.
.
program statement
X-OFF (S^C)
carriage return
linefeed
X-OFF (S^C)
tape trailer

C Punching programs to paper tape off-line

Programs may be punched off-line using the same output sequence that the PUNch or XPUnch command would create on-line.

1. The terminal LOCAL/OFF/LINE switch is turned to LOCAL.
2. The user is not logged on.
   a. Turn the paper tape punch to the ON position.
   b. Type R^C.
   c. Depress the SHIFT key, the CONTROL key and the REPEAT key, and then depress the letter P key. This causes a series of NULLS to be punched on the tape as a leader to make loading the tape on the reader easier.
   d. Type program statement.
   e. For the equivalent of an XPUnched tape, type S^C. (For the equivalent of a PUNched tape, omit this step.)
   f. Type carriage return.

g. Type linefeed.

h. Repeat steps d. through g. for each line of the program.

i. Repeat step c.

j. Type T^C.

k. Turn the paper tape punch to the OFF position.

Programs punched off-line using the XPUnch format can be read by an executing program only on terminals equipped with the X-ON, X-OFF feature.

D Punching data to paper tape off-line

Data are punched to paper tape off-line using the same output sequence that the XPUnch command would create on-line for a program.

1. The terminal LOCAL/OFF/LINE switch is turned to LOCAL.
2. The user is not logged on.

   a. Turn the paper tape punch to the ON position.

   b. Type R^C.

   c. Depress the SHIFT key, the CONTROL key and the REPEAT key, and then depress the letter P key. This causes a series of NULLS to be punched on the tape as a leader to make loading the tape on the reader easier.

   d. Type line of data, including comma separators where appropriate.

   e. Type S^C.

   f. Type carriage return.

   g. Type linefeed.

   h. Repeat steps d. through g. for each line of data.

   i. Repeat step c.

   j. Type T^C.

   k. Turn the paper tape punch to the OFF position.

III Reading punched paper tape into the computer

The TAPe command is used to read a program into the computer from punched paper tape, and the KEY command removes the computer from tape mode.

The user is reminded that the computer views program entry from paper tape mode just as it would program entry from the terminal keyboard. In particular, this means that while erroneous program statements will generate error messages, they will *not* be included in the

computer's copy of the program. This is identical to the way the computer deals with erroneous program statements typed in from the keyboard. In order for these statements to be included in the computer's copy of the program, they must be correctly re-entered, either from the keyboard, or from additional paper tape.

A TAPe

The TAPe command alerts the computer that the next input will be from paper tape. It is commonly used to read programs in PUNch format and not data from paper tape.

1. The terminal LOCAL/OFF/LINE switch is turned to LINE.
2. The user is logged on.
3. The user's work area is clear.
4. The paper tape to be read is located into the tape reader.
    a. Type TAP and strike the RETURN key.
    b. Turn the paper tape reader to the ON position.
    c. The tape is read and echoed at the terminal.
    d. After the tape is read, turn the paper tape reader to the OFF position.

While the computer is in tape mode, error messages are suppressed. Input of any command causes termination of tape mode, and output of waiting error messages. The KEY command is commonly used for this purpose.

B KEY

The KEY command alerts the computer that the next input will be from the terminal keyboard, thus:

1. Type KEY and strike RETURN

If error messages were generated from entry of the program tape, they will be printed. If no error messages were waiting, then there will be no output, and the computer will be ready for the next command, usually a SAVe command or a RUN command.

C Inputting data from paper tape to an executing program

To read a data tape into an executing program, the terminal must be equipped with the X-ON, X-OFF feature. The procedure is:

1. Terminal LOCAL/OFF/LINE switch is turned to LINE.
2. The user is logged on.

3. The executing program contains one or more INPUT statements.

   a. If the INPUT statement is of the form

      0013 INPUT item,item,item

      then the ? symbol and an X-ON character (Q^C) will be output automatically. Data are read from the tape until an X-OFF character (S^C) is encountered. (The user is reminded that a string of more than 254 characters will generate a terminal error.)

   b. If the INPUT statement is of the form

      0013 INPUT "string",item,item,item

      then the last character in "string" must be an X-ON (Q^C) in order to activate the tape reader. "string" will be output, and then data are read from the tape in the same way as from the INPUT statement form under item 3.a. above.

Appendix E

PUBLIC LIBRARY UTILITY PROGRAMS

The System public library contains over 100 programs which are available to the user. The programs discussed in this Appendix are those which bear some relation to other material covered in this Manual, as for example, the LIST utility program to the LISt command. The entire contents of the System public library is cataloged and described in the Public library index manual.

1. CATALOG (GET-$$$CATALOG)

Outputs the user's library catalog to a (pre-existing) file or to the terminal. If output to a file, each catalog entry (file or program) is a 43-character string which contains the following information:

CHARACTERS	DATA
1 — 5	Serial Number
6 — 15	Name
16 — 18	Code
19 — 23	Length
24 — 28	Sectors
29 — 33	Saved Date
34 — 38	Accessed Date
39 — 43	Uses

There are 5 catalog entries per file record, beginning in record 2. The first record contains the date that the catalog was written to the file. For example, the following library catalog is written to a file named CATFILE:

```
CAT
 S/N     NAME         CODE   LEN.   SECT. SAVED    ACCESS    USE

00265   CATFILE        F    00007   0008  138/76   138/76   00002
00258   PORTS          P    00064   0001  132/76   132/76   00001
00253   PROG           P    00366   0003  132/76   132/76   00000
00252   FILE           F    00010   0011  132/76   135/76   00024
00251   WRITE          F    00005   0006  132/76   132/76   00008
00247   WRITE/EXEC     N    00050   0001  132/76   132/76   00001
00246   EXEC           F    00010   0011  132/76   132/76   00003
00245   VARS           P    00111   0001  131/76   131/76   00000
00244   PROG1          P    00265   0003  127/76   127/76   00000
00243   Y              P    00047   0001  127/76   127/76   00000
00241   X              P    00215   0002  127/76   127/76   00000
00234   PTUSG3         P    00034   0001  127/76   127/76   00000
00231   TESTP          N    00178   0002  125/76   125/76   00003
00229   MYPROG         P    00172   0002  124/76   138/76   00006
```

```
00228   REF           P    00175   0002   124/76   124/76   00000
00227   START         P    00138   0002   124/76   124/76   00000
00222   FILEUPDATE    P    00233   0002   121/76   121/76   00003
00202   TIMEOUT       P    00138   0002   112/76   119/76   00002
00089   EVERYONE     7N7   00003   0001   191/75   191/75   00000
00088   PER@005       P4   00003   0001   191/75   191/75   00000
00087   PRO@005      4P    00003   0001   191/75   209/75   00001
00086   PER@002       P2   00003   0001   191/75   209/75   00001
00085   PRO@002      2N    00003   0001   191/75   209/75   00001
00084   PERMASTER     P1   00003   0001   191/75   209/75   00003
00083   PROMASTER    1N    00003   0001   191/75   209/75   00002

TOTAL STORAGE = 00068 SECTORS
```

```
GET-$$$CATALOG
RUN
CATALOG

ACCT #, HI S/N, LOW S/N ?M050
OUTPUT (T-TERM. F-FILE ) ?F
FULL FILE NAME ? CATFILE

DONE
```

The user may output only a portion of his library catalog if he wishes, by specifying a range of catalog entry numbers following input of the account name and separated from it by commas, thus:

ACCT #, HI S/N, LOW S/N?M050,244,202

2. F-LIST (GET-$$$F-LIST)

The contents of any unscreened file can be listed by the System public library utility program named F-LIST. The following example lists the contents of records 1 through 4 of the user's file named CATFILE.

```
GET-$$$F-LIST
RUN
F-LIST

FILE NAME ?CATFILE
FILE LENGTH = 7


1ST R#?1
END R#?4

RECORD: 0001
 76139.
'EOR'   [002 WORDS]
```

```
RECORD: 0002
00265CATFILE    0F000007000081387613976000004   00258PORTS      0P0000640000
1132761327600001   00253PROG       0P000366000031327613276000000   00252FILE
      0F000010000111327613576000024   00251WRITE      0F000005000061327613276
7600008
'EOR'   [115 WORDS]


RECORD: 0003
00247WRITE/EXEC0N000050000011327613276000001   00246EXEC       0F0000100001
1132761327600003   00245VARS       0P000111000011317613176000000   00244PROG
1      0P000265000031277612776000000   00243Y          0P000047000011277612776
7600000
'EOR'   [115 WORDS]


RECORD: 0004
00241X          0P000215000021277612776000000   00234PTUSG3     0P0000340000
1127761277600000   00231TESTP      0N000178000021257612576000003   00229MYPR
OG     0P000172000021247613876000006   00228REF        0P000175000021247612476
7600000
'EOR'   [115 WORDS]


TOTALS:   RECORDS 1     THRU 4

  NUMERICS 1
   STRINGS 15
     WORDS 347

DONE
```

3. LIST (GET-$$$LIST)

Outputs the list of a program to a file or to the terminal. The user specifies the program to be listed, the file to contain the listing and other information, through a series of name-listed parameters, as follows:

PARAMETER	NAME	USE
output list to the terminal	T	T
output list to a file	F	F=filename
lowest statement number to be listed	L	L=number
highest statement number to be listed	H	H=number
program to be listed	P	P=programname

The parameters are entered in response to the word OPTIONS and are separated by commas. For example, to list a portion of the following program to a file named LISTFILE as well as to the terminal:

```
GET-PROG
LIST
PROG

0010  DIM A$[81],B[329],N$[254],Q$[10]
0020  DIM K$[3]
0030  DIM L$[254]
0040  DIM C$[254]
0050  MAT B=ZER
0060  A=0
0070  FILE #1,1;"#W904,SCH.SRT983"
0080  Q$="0123456789"
0090  N$=".003048780487804878"
0100  ON  ERROR  THEN 300
0110  ON  END #1 THEN 300
0120  READ #1;A$
0130  A=A+1
0140  L$=""
0150  H=1
0160   FOR K=9 TO 33
0170   T=ASC(A$[K:K])
0180   GOSUB 350
0190   NEXT K
0200  C$=L$*N$
0210   FOR J=1 TO 254
0220   IF C$[J:J]="." THEN 240
0230   NEXT J
0240  C$=C$[J:LEN(C$)]
0250  C$=C$*"329"
0260  C$=C$+"1"
0270  R=INT(VAL(C$))
0280  B[R]=B[R]+1
0290  GOTO 120
0300  PRINT A
0310   FOR K=1 TO 329
0320   PRINT K,B[K]
0330   NEXT K
0340  END
0350  F=1
0360  K$="000"
0370  S=T-INT(T/10)*10
0380  K$[F:F]=Q$[S+1:S+1]
0390  F=F+1
0400  T=INT(T/10)
0410  IF T#0 THEN 370
0420   FOR G=3 TO 1 STEP -1
0430   L$[H:H]=K$[G:G]
0440   H=H+1
0450   NEXT G
0460  RETURN
0470  END
```

```
OPE-LISTFILE,10
GET-$$$LIST
RUN
LIST

OPTIONS ? P=PROG,T,F=LISTFILE,L=160,H=340
   BTI-4000    R.05    18-MAY-76      PROGRAM- PROG


 0160    FOR K=9 TO 33
 0170    T=ASC(A$[K:K])
 0180    GOSUB 350
 0190    NEXT K
 0200   C$=L$*N$
 0210    FOR J=1 TO 254
 0220    IF C$[J:J]="." THEN 240
 0230    NEXT J
 0240   C$=C$[J:LEN(C$)]
 0250   C$=C$*"329"
 0260   C$=C$+"1"
 0270   R=INT(VAL(C$))
 0280   B[R]=B[R]+1
 0290   GOTO 120
 0300   PRINT A
 0310    FOR K=1 TO 329
 0320    PRINT K,B[K]
 0330    NEXT K
 0340   END

SOURCE LENGTH   366
```

The parameters may be listed in any order, and all are optional except,

(a) the name of the program to be listed

(b) a destination for the output list (file or terminal — both may be specified, if desired)

All programs to which the user has access may be referenced by the LIST program except those with a RUN ONLY screen. (Screens are discussed in Appendix A, the CATalog command.) If the user attempts to LIST a RUN ONLY program, the message

RUN ONLY

will be printed and the LIST program will terminate.

The file into which the referenced program is to be listed must be created before the LIST program is run, and must be available to the user on a READ/WRITE basis. If the file is unavailable or does not exist, the error message

FILE NOT THERE OR NOT SHARED

will be printed and the LIST program will terminate.

The F-LIST utility may be used to display the contents of LISTFILE:

```
DONE


GET-$$$F-LIST
RUN
F-LIST

FILE NAME ?LISTFILE
FILE LENGTH = 10


1ST R#?1
END R#?3

RECORD: 0001
 NAM-PROG
'EOR'  [006 WORDS]



RECORD: 0002
 0160   FOR K=9 TO 33   0170   T=ASC(A$[K:K])   0180   GOSUB 350   0190
  NEXT K   0200  C$=L$*N$   0210   FOR J=1 TO 254   0220   IF C$[J:J]=".
" THEN 240   0230   NEXT J   0240  C$=C$[J:LEN(C$)]   0250  C$=C$*"329"
  0260  C$=C$+"1"
'EOR'  [122 WORDS]



RECORD: 0003
 0270  R=INT(VAL(C$))   0280  B[R]=B[R]+1   0290  GOTO 120   0300  PRINT
 A   0310   FOR K=1 TO 329   0320   PRINT K,B[K]   0330   NEXT K   0340
 END  1  1
'EOF'  [080 WORDS]


TOTALS:  RECORDS 1    THRU 3

  NUMERICS 0
   STRINGS 22
     WORDS 208

DONE
```

The LIST utility stores the program code in a format such that the filed program can be modified by the System public library utility named EDIT. (See the BTI Public Library Manual for details.)

4. XREF (GET-$$$XREF)

The references in a program (i.e., variable names, functions, etc.) may be broken out and displayed by the System public library utility program named XREF. The program to be cross referenced must reside in the user's library and must be completely unprotected, that is it must appear in the user's library catalog as a "P" entry (see Appendix A, the CATalog command). The program cannot be killed and re-saved, nor can it be redesignated during execution of XREF.

The program named MYPROG,

```
LIS
MYPROG

0010  DIM A$[254],#1[5]
0020  DEF FNA(X)=1+INT((N+1)/2)
0030  FILE #1,1;"MYFILE"
0040  Z=1
0050  X=TYP(1)
0060  IF X=3 THEN 200
0070  IF X=1 THEN 130
0080  READ #1,Z;A$
0090  N=LEN(A$)
0100  PRINT "RECORD ";Z;" CONTAINS A STRING OF ";N
0110  PRINT "CHARACTERS, REQUIRING ";FNA(N);" WORDS OF STORAGE."
0120  PRINT
0130  Z=Z+1
0140  GOTO 50
0200  PRINT "END OF FILE"
9999  END
```

may be cross referenced as follows:

```
GET-$$$XREF
RUN
XREF

05/04/76   0759
FUNCTIONS/STATEMENTS?Y
VARIABLES/LINE NUMBERS?Y
CONSTANTS?Y
ENTER PROGRAM NAME: MYPROG
```

```
(1) PREDEFINED FUNCTIONS IN MYPROG

   INT                20
   LEN                90
   TYP                50

(2) STATEMENTS ET. AL. IN MYPROG

   DEF                20
   DIM                10
   END              9999
   FILE               30
   #(FILE)            10     30     80
   GOTO              140
   IF                 60     70
   (LET)              40     50     90    130
   PRINT             100    110    120    200
   READ               80

(3) CROSS-REFERENCE OF MYPROG

     A$        10     80     90
   FNA         20    110
     N         20     90    100    110
     X         50     60     70
     Z         40     80    100    130    130

(4) 0050      140
   0130       70
   0200       60

(5) CONSTANTS IN MYPROG

    1                 20     20     30     30     40     50     70     80    130
    2                 20
    3                 60

   DONE
```

In the example above, the pre-defined (intrinsic) functions in the program are listed first (in Group (1)), alphabetically with the numbers of the lines in which they are used. The statement types used are listed in Group (2), again alphabetically, with their line numbers in the program, in ascending numerical order. Group (3) are the program's user-defined functions and variables, in alphabetical order with their line numbers in ascending numerical order, and group (4) are the line number references such as in IF . . . THEN statements and GOTO statements. The left-most column of numbers in group (4) are the destination line numbers and the right hand column(s) are the source line numbers. The last group of numbers are the numeric constants in the program, with the numbers of the lines in which they are used.

INDEX

For more information and assistance contact:

East: 3 Executive Campus, Cherry Hill, NJ 08002 (609) 662-1122
South: 1545 W. Mockingbird Lane, Suite 5013, Dallas, TX 75235 (214) 630-2431
Midwest: 2850 Metro Drive, Minneapolis, MN 55420 (612) 854-1122
West: 870 West Maude Avenue, Sunnyvale, CA 94086 (408) 733-1122

Basic Timesharing Inc.

870 West Maude Avenue, Sunnyvale, California 94086 (408) 733-1122

Printed in U.S.A.

1020-R0177