PICK BASIC
reference manual

88A00778A02

ZEBRA
FAMILY

GENICOM AUTOMATION

RECORD OF REVISIONS

Title: PICK BASIC Reference Manual

Document No. 88A00778A02

Date	Revision Record
Mar 84	Original Issue
Feb 85	Revision A02 - Change Package (85A00517A01)

PICK BASIC
reference manual

88A00778A02

1045 South East Street P.O. Box 4883
Anaheim, California 92803
(714)778-4800 (800)854-6234
TWX 910-591-1695 TELEX 685-513

RECORD OF REVISIONS

Title: PICK BASIC Reference Manual

Document No. 88A00778A02

Date	Revision Record
Mar 84	Original Issue
Feb 85	Revision A02 - Change Package (85A00517A01)

FOREWORD

This document is one of a family of ZEBRA reference manuals devoted to PICK processors that are on call within the PICK operating system. Before reading this document and using the processor described, it is recommended that you first become familiar with the PICK terminal control language and file structure. These subjects are thoroughly covered in 88A00782A, listed below with other documents covering PICK processors.

Document No.	Title
88A00757A	PICK Operator Guide
88A00758A	ACCU-PLOT Operator Guide
88A00759A	COMPU-SHEET Operator Guide
88A00760A	Quick Guide for the PICK Operating System
88A00774A	PICK Utilities Guide
88A00776A	PICK ACCESS Reference Manual
88A00777A	PICK SPOOLER Reference Manual
88A00779A	PICK EDITOR Referance Manual
88A00780A	PICK PROC Reference Manual
88A00781A	PICK RUNOFF Reference Manual
88A00782A	Introduction to PICK TCL and FILE STRUCTURE
88A00783A	PICK JET Word Processor Guide

TABLE OF CONTENTS

introduction 1

1.1 THE BASIC LANGUAGE

BASIC (Beginners All-Purpose Symbolic Instruction Code) is a simple yet versatile programming language suitable for expressing a wide range of problems. Developed at Dartmouth College in 1963, BASIC is a language especially easy for the beginning programmer to master.

The extended PICK BASIC has the following features:

- Optional statement labels (statement numbers)
- Statement labels of any length
- Multiple statements on one line
- Computed GOTO statements
- Complex IF statements
- Multi-line IF statements
- Priority case statement selection
- String handling with variable length strings up to 32,267 characters
- External subroutine calls
- Direct and indirect calls
- Magnetic tape input and output
- Fixed point arithmetic with up to 15 digit precision
- ACCESS data conversion capabilities
- PICK access and update capabilities
- File level or group level lock capabilities
- Pattern matching
- Dynamic arrays

Table 1-1 lists the BASIC statements. The BASIC intrinsic functions are listed in Table 1-2.

Table 1-1. BASIC Statements

ABORT	END	INPUT	MATWRITEU	PROMPT	RQM
BREAK	ENTER	INPUT@	NEXT	READ	SELECT
CALL	EQUATE	INPUTERR	NULL	READNEXT	SLEEP
CASE	EXECUTE	INPUTNULL	ON GOSUB	READU	STOP
CHAIN	FOOTING	INPUTTRAP	ON GOTO	READT	SUBROUTINE
CLEAR	FOR	LOCATE	OPEN	READV	UNLOCK
CLEARFILE	GO	LOCK	PAGE	READVU	WEOF
COMMON	GOSUB	LOOP	PRECISION	RELEASE	WRITE
CRT	GOTO	MAT	PRINT	REM	WRITEU
DATA	GO TO	MATREAD	PRINTER	RETURN	WRITET
DELETE	HEADING	MATREADU	PROCREAD	RETURN TO	WRITEV
DIM	IF	MATWRITE	PROCWRITE	REWIND	WRITEVU
ECHO					

Table 1-2. BASIC Intrinsic Functions

@	COUNT	ICONV	NUM	SPACE
ABS	DATE	INDEX	OCONV	SQRT
ALPHA	DCOUNT	INSERT	PWR	STR
ASCII	DELETE	INT	REM	SYSTEM
CHAR	EBCDIC	LEN	REPLACE	TAN
COL1	EXP	LN	RND	TIME
COL2	EXTRACT	MOD	SEQ	TIMEDATE
COS	FIELD	NOT	SIN	TRIM

1.2 BASIC FILE STRUCTURE

A fixed structure is established for BASIC source files. The file MUST have a dictionary and a separate data level. The BASIC source programs are stored in the data level of the file. The compiler writes the object and the symbol file as one record into the dictionary. This makes it much simpler to manipulate the program source. It can be LISTed, T-DUMPed, T-LOADed, and so on, without having to select the source items. The object record has the same format as a pointer-file record and so the dictionary "D" pointer must have a "DC" in attribute one. The advantages of this format are:

1. The object can be protected with access/update locks.

2. The object saves/restores with the account on account-saves.

3. The CATALOG function is not necessary for run time efficiency.

4. BASIC Debug can tell the name of the item and verify the object code integrity.

1.2.1 CREATING BASIC FILES: CREATE-PFILE

The CREATE-PFILE verb should be used to create your BASIC files and dictionaries. CREATE-PFILE performs in the same manner as CREATE-FILE except that it automatically places a "DC" in attribute 1 of the dictionary. The general form of CREATE-PFILE is:

```
CREATE-PFILE file-name m1{,s1} m2{,s2}
   or        dict-name,data-name  m1{,s1} m2{,s2}
   or        DICT file-name  m1{,s1}
```

This will create a file for the file-name specified and its associated dictionary. The modulo and separation values for the dictionary are given first (m1,s1) and the values for the file lost (m2,s2). If s is not specified, s=1 is assumed.

The form dict-name,data-name must be used if file-name describes one of multiple files which use the same dictionary.

A dictionary may be created without a data file by using the DICT form shown.

1.3 BASIC PROGRAMS

1.3.1 BASIC STATEMENTS

A BASIC program consists of a sequence of BASIC statements optionally terminated by an END statement. BASIC statements begin with a keyword which performs a specific task. For example:

```
WRITE 100*5 ON "RATE"
```

More than one BASIC statement may appear on the same program line if the statements are separated by semicolons. For example:

```
PRINT I; GOTO 50; *PRINT THE VALUE OF I
```

Any BASIC statement may begin with a statement label. The label is used so that a statement may be referenced by another statement in the program. A statement label may be any whole number. If may not be any alphabetic character or combination of alphabetic characters. The use of a label is optional.

BASIC statements may contain arithmetic, relational, and logical expressions. These expressions are formed by combining specific operators with variables, constants, or BASIC Intrinsic Functions. The value of a variable may change dynamically throughout the execution of the program. A constant, as its name implies, has the same value throughout the execution of the program. An Intrinsic Function performs a predefined operation upon the parameter(s) supplied.

Except where specifically prohibited (which will be shown in the following sections of this manual) blank spaces may appear in the program line. You should use them freely throughout the program to enhance the appearance of the program.

1.3.2 COMMENTS WITHIN THE BASIC PROGRAM

A helpful feature when writing BASIC programs is the REMARK statement. By using a REM (REMARK) statement, you may place comments anywhere in the program without affecting program executing.

A REMARK statement may be specified in three ways:

1. By the letters REM.
2. By an asterisk (*).
3. By an exclamation point (!).

Any of these, when placed at the beginning of a statement, will allow you to use any combination of characters up until the end of the line. In this way, you may explain or document your program to any extent you wish. There is no limit to the number of comment lines you may use. Brief comments may be appended to or may prefix BASIC program lines.

A sample BASIC program with and without the use of the REMARK statement is given below:

```
    I=1
  5 PRINT I
    IF I=10 THEN STOP
    I = I+1
    GOTO 5
    END

REM PROGRAM TO PRINT THE NUMBERS
 *  FROM ONE TO TEN
 *
    I=1; * START WITH ONE
  5 PRINT I; * PRINT THE VALUE
    IF I=10 THEN STOP; * STOP IF FINISHED
    I = I+1; * INCREMENT I
    GOTO 5; * START OVER
    END
```

1.3.3 SPECIAL PICK USE OF BASIC

A BASIC program, when stored, is a file item. It is referenced by its item-name (or item-id) which is the name it is given when created by the EDITOR. An individual line within the BASIC program is an attribute.

1.4 CREATING AND COMPILING WITH BASIC

A BASIC program is created like any other data-file item by using the EDITOR. Once this source code item has been filed, it is compiled by issuing the COMPILE or the BASIC verb. To enter the EDITOR, you issue the following verb:

```
ED (or EDIT) file-name item-id
```

The EDITOR processor will then be entered, and you may begin entering your BASIC program. For ease of instruction indentation, you may use the FORMAT verb (see Section 1.4.1) or set tab stops (either at the TCL level or while the EDITOR processor is in control).

The program name is specified by 'item-id' and the program is to be stored in the file specified by 'file-name'. Users will typically have a file exclusively devoted to the storage of BASIC programs. The BASIC compiler stores the object code in the same file, but not in the same item as the source code.

Once the BASIC program has been entered and filed, it may be compiled by issuing the COMPILE verb. COMPILE is a TCL-II verb which creates two new file items: one contains the compiled BASIC program (the object code), and the other contains a symbol definition table of the variables used in the program. All three items (source, object and symbol table) are stored in the same file. The COMPILE command format is:

```
COMPILE file-name item-list {(options)}
 or
BASIC   file-name item-list {(options)}
```

The file-name is the name of the file containing the BASIC program(s). The 'item-list' consists of one or more item-id's (program names) separated by one or more blanks. The 'options' parameter is optional and if used, must be enclosed in parentheses. Multiple options should be separated by commas. Valid options are listed below. Detailed descriptions of each are provided in the following section.

A	Assembled code option
C	Suppress End of Line (EOL) opcodes from object code
E	List error lines only
L	List BASIC program
M	List map of BASIC program
P	Print compilation output on line printer
S	Suppress generation of symbol table
X	Cross reference all variables

An example of a BASIC program ("COUNT") that is originated via EDIT, then filed and compiled:

```
* >TABS I 4,8,12  [CR]   <----------  User sets input tabs at TCL level

* >ED BP COUNT  [CR]     <----------  User edits item 'COUNT' in file 'BP'
                                      (BASIC Programs)

 NEW ITEM
 TOP
*  .I  [CR]             <----------  User enters input mode and begins
                                      to enter program

* 001* PROGRAM COUNTS FROM 1-10 *  [CR]
* 002    FOR I = 1 TO 10  [CR]  <----  Entered with [C] I (or TAB key)
* 003      PRINT I  [CR]  <------      depressed once for indentation
* 004    NEXT I  [CR]           |      to first tab stop.
* 005  END  [CR]                |
* 006   [CR]                    -----  [C] I (or TAB key) depressed
TOP                                   twice for second tab stop
                                      indentation
* .FI  [CR]            <-------
                              |
                              ------  User files item
'COUNT' FILED

* >COMPILE BP COUNT  [CR]  <---------  User issues compile command
  *****
 PROGRAM 'COUNT' COMPILED!  n FRAMES USED.
```

1.4.1 FORMAT VERB

You may use the FORMAT verb to automatically indent IF...THEN...ELSE, FOR...NEXT, LOOP...DO...REPEAT and CASE statements. It has the general form:

```
FORMAT
```

There will be a prompt BASIC FILE NAME? for you to enter the name of the file storing your BASIC program. You will then be prompted:

```
OUTPUT TO S(CREEN), P(RINTER) OR N(O)?
```

Enter S, P, or N for formatted output to terminal, printer, or no output, respectively. You will then be asked:

```
BASIC PROGRAM NAME?
```

After the program name is entered, your output will be processed as specified. Note that FORMAT may be used both before and after a COMPILE.

1.4.2 BASIC COMPILER OPTIONS

This section describes the options available when issuing the BASIC or COMPILE compile statement. The options parameter must be enclosed in parentheses with individual options separated by commas.

Options	Description
A	Assembled code. Generates a listing of the source code line numbers, the labels and the BASIC opcodes used by the program. This is a 'pseudo' Assembly code listing which allows the user to see what BASIC opcodes the program has generated. The hexadecimal numbers on the left of the listing are the BASIC opcodes and the mnemonics are listed on the right. See the example for the assembled code listing of the BASIC program "COUNT" (from Section 1.4).
C	Catalog. Suppresses the end-of-line (EOL) opcodes from the object code item. The EOL opcodes are used to count lines for error messages. This eliminates 1 byte from the run time object code for every line in the source code. This option is designed to be used with debugged cataloged programs. Any run-time error message will specify a line number of 1.
E	List Error lines only. Generates a listing of the error lines encountered during the compilation of the program. The listing indicates line number in the source code item, the source line itself and a description of the error associated with the line.
L	List program. Generates a line by line listing of the program during compilation. Error lines with associated error messages are indicated.
M	Map. Generates a variable map and a statement map, both of which are printed out after compilation. These maps show where the program data has been stored in the user's workspace. The variable map lists the offset in decimal (from the beginning of the seventh frame of the IS buffer) of every BASIC variable in the program, i.e., the form: 20 xxx 30 yyy shows that the descriptor of variable 'xxx' starts on byte 20 and the descriptor of variable 'yyy' starts on byte 30 of the seventh frame of the IS buffer. Descriptors are 10 bytes in length. The statement map shows which statements of the BASIC program are contained in the frames of the OS buffer. If the program is run, frame number '01' refers to the seventh frame of the OS buffer. If the program is cataloged, frame 01 will be specified in the catalog pointer item in the POINTER-FILE. The statement map may be used to determine if frequently executed loops cross frame boundaries.
P	Printer. Routes all output generated by the compilation to the printer.

S Suppress symbol table. Suppresses the symbol table item which is normally generated during compilation. The symbol table item is used exclusively by the BASIC Debugger for reference, therefore, it must be kept on file only if the user wishes to use the Debugger.

X Cross reference. Creates a cross reference of all the labels and variables used in a BASIC program and stores this information in the BSYM file. A BSYM file must exist (a modulo and separation of 1,1 should be sufficient). The "X" option first clears the BSYM file information in the BSYM file then creates an item for every variable and label used in the program. The item-id is the variable or label name. The attributes contain the line numbers of where the variable or label is referenced. An asterisk will precede the line number where a label is defined, or where the value of the variable is changed. No output is generated by this option. An attribute definition item should be placed in the dictionary of the "BSYM" file which allows a cross reference listing of the program to be generated by the command:

```
>SORT BSYM BY LINE-NUMBER LINE-NUMBER
```

An example of a BASIC A option listing for "COUNT":

```
Source      BASIC       Pseudo
Code        Object      Assembly
Line No     Code        Code

001         01          EOL
002         03          LOADA       I
002         FD          LOAD.       1
002         20          ONE
002         2D          SUBTRACT
002         5F          STORE
002             *1001
002         05          LOADN       10
002         03          LOADA       I
002         20          ONE
002         28          FORTEST     *2001
002         01          EOL
003         5D          LOAD        I
003         50          PRINTCRLF
003         01          EOL
004         06          BRANCH      *1001
004             *2001
004         01          EOL
005         01          EOL
006         45          EXIT

        [BO] LINE 6 COMPILATION COMPLETED
```

1.5 EXECUTING BASIC PROGRAMS

A compiled BASIC program is executed by issuing the RUN verb. The command format is:

```
RUN file-name item-id {(options)}
```

The "file-name" gives the name of the file where the compiled BASIC program is stored and "item-id" specifies the name of the program to be executed. The "options" parameter is optional. If used, it must be enclosed in parentheses with multiple options separated by commas. Valid options are given below:

Options	Description
A	Abort. Inhibits entry to the BASIC Debugger under all error conditions. Instead, the program will print a message and terminate execution.
D	Run-time Debug. Causes the BASIC debugger to be entered before the start of program execution. The BASIC debugger may also be called at any time while the program is executing by pressing the BREAK key on the terminal.
E	Errors. Forces the BASIC run time package to enter the BASIC Debugger whenever an error condition occurs. The use of this option will force the operator to either accept the error by using the Debugger or exit to TCL.
I	Inhibit initialization of data area. The "I" option should only be used in the "CHAIN" statement. (Refer to BASIC CHAIN statement.)
N	Nopage. Cancels the default wait at the end of each page of output.
P	Printer on. This has the same effect as a BASIC PRINTER ON statement; directs all program output to the printer.
S	Suppress. Suppresses run-time warning messages.

An example of the execution of sample BASIC program:

```
* >RUN PROGRAMS TEST  [CR]

THIS IS

A TEST
```

1.5.1 RUNNING BASIC PROGRAMS FROM A PROC

A BASIC program may be run from a PROC. The following example illustrates the use of a BASIC program in conjunction with the ACCESS SSELECT verb. A PROC named LISTBT is written as follows:

```
PQ
HSSELECT BASIC/TEST
STON
HRUN BP LISTIDS
P
```

A BASIC program named LISTIDS is written as follows:

```
    OPEN '','BASIC/TEST' ELSE PRINT 'FILE MISSING'; STOP
10  N = 0
20  READNEXT ID ELSE STOP
    PRINT ID 'L################':
    N = N + 1
    IF N>= 4 THEN PRINT; GOTO 10
    GOTO 20
    END
```

By typing in the following:

```
LISTBT
```

at the TCL level, the PROC LISTBT selects the item-id's contained in the file BASIC/TEST and invokes the BASIC program LISTIDS to list the item-id's selected, four to a line.

For further information, refer to the PROC Manual.

1.5.2 ALTERING EXECUTION TIME: RQM AND SLEEP STATEMENTS

The RQM (Release Quantum) statement terminates the executing program's current quantum (time-slice). The RQM statement may be used to alter program execution speed. The SLEEP statement follows the same form and performs the same functions as the RQM statement. The general form of the RQM and SLEEP statements:

```
RQM {time}
SLEEP {time}
```

where time is either an integer specifying the number of seconds to sleep or is in the format "hh:mm:ss" specifying the time until which to sleep. Note that the "hh:mm:ss" format is a 24-hour military time, and requires the use of quotes.

The time-shared environment of the PICK system allows several programs to execute together with each program executing for a specific time period (called a time-slice or quantum) and then pausing while another program continues execution. Since the RQM and SLEEP statements terminate a program's current time-slice, they may be used in time-consuming program loops to allow increased execution speed to other concurrently executing programs. They may also be used to cause pauses in program execution.

Examples of the use of RQM:

Correct Use	Explanation
*PROGRAM SEGMENT TO SOUND *TERMINAL "BELL" FIVE TIMES. * BELL=CHAR(7) FOR I=1 TO 5 PRINT BELL: RQM NEXT I END	RQM statement allows enough time for bell to be heard as discrete "beeps".

Incorrect Use	Explanation
X1='TEST' RQM X1	Only a numeric time parameter is allowed with the RQM statement.

1.6 PROGRAM SHARING

BASIC programs may be shared by users on various accounts by creating a Q-pointer to the BASIC program file in the users' accounts.

1.6.1 CATALOG VERB

A BASIC program may be evoked at the TCL level by the creation of a catalogued program. This is done by using the CATALOG instruction. The verb has the following general form:

```
CATALOG file-name item-id
```

The "file-name" refers to the BASIC program file which stores the program and "item-id" specifies the previously compiled BASIC program which is to be accessed from TCL. If there are no conflicts, the system will respond with:

```
'Item-id' CATALOGED
```

Once a program is cataloged, it is 'run' simply by issuing the program name at the TCL prompt. The TCL-II verb which is added to the user's Master Dictionary (if not already present) has the following form:

```
1)  P
2)  E6
3)
4)
5)  XXXXX
```

where XXXXX is the BASIC program file name. If an item already exists in the user's Master Dictionary which is not in the above form, the system will respond with:

```
[415] ITEM 'Item-id'  EXISTS ON FILE
```

and the program will not be cataloged.

1.6.2 DECATALOG VERB

In order to delete the object code of a BASIC program, the DECATALOG verb has been provided, which has the following form:

```
DECATALOG  file-name item-id
```

where "file-name" is the name of the file containing the BASIC proram and "item-id" is the name of the program.

An item is maintained in the dictionary of the BASIC program file for each compiled BASIC program. The DECATALOG statement will delete that item from the dictionary, return all of the overflow space, and delete the BASIC program name verb from the user's Master Dictionary. Note that it will not delete the program from the file, so it may be recompiled and recataloged at a later time.

After deletion, verbs executed from other accounts will receive the message:

```
'Item-id'  NOT ON FILE
```

The CATALOG and DECATALOG verbs are TCL-II verbs. This means that they require the BASIC program file-name and one or more explicit item-ids, or a '*' (meaning all), or that a list be in effect. It also means that you can catalog all of the BASIC programs in one file by using the CATALOG verb only once, and decatalog them similarly with the DECATALOG verb.

The effect of the CATALOG verb is to point to the file which contains the pointer to the object code.

The primary purpose of the DECATALOG verb is to remove the object code string from the system. This string is pointed to by the pointer record in the dictionary of the BASIC program file where the proram resides. The program does not need to be cataloged in order to use the DECATALOG verb. It will also remove the pointer left in the Master Dictionary by the CATALOG verb if there is one.

Note that with both of these verbs, the BASIC program file name and the program name are the only parameters used by the system. The BASIC program name is transferred to the object code pointer as its name, and to the Master Dictionary pointer to the dictionary of the BASIC file. Similarly, the object pointer will reside in the dictionary of the file which contains the source program in the data section, and the pointer which results from the CATALOG verb will point to that file.

data representation 2

2.1 CONSTANTS AND VARIABLES

There are two types of data within PICK; numeric and string. These data types are represented within the BASIC program as either constants or as variables.

Numeric data consists of a series of digits and represents an amount, such as 255. String data consists of a set of characters, for example, a name and address:

```
JOE DOE, 430 MAIN, ATOWN, CA.
```

Both numeric and string data types may be represented within the BASIC program as either constants or variables. A constant, as its name implies, has the same value throughout the execution of the program. A numeric constant may contain up to 15 digits, including a maximum of 4 digits following the decimal point and must be in the range:

```
-14,073,748,835.0000 to 14,073,748,835.0000
```

If the precision (see Section 2.5, Selecting Numeric Precision) of the program is 4 digits; by setting the PRECISION to a value less than 4, the range of the allowable numbers is increased accordingly.

The unary minus sign is used to specify negative constants. For example:

```
-17000000
-14.3375
```

A string constant is represented by a set of characters enclosed in single quotes, double quotes, or backslashes. For example:

```
"THIS IS A STRING" 'ABCD1234#*' \HELLO\
```

If any of the string delimiters (', " or \) are to be part of the string, then one of the other delimiters must be used to delimit the string. For example:

```
"THIS IS A 'STRING' EXAMPLE"
\THIS IS A "STRING" EXAMPLE\
```

A string may contain from 0 to 32,267 characters, which is the maximum length of a PICK file item.

A number of valid and invalid string constants examples:

Valid String	Invalid String
"ABC%123#*4AB"	ABC123 (quotes are missing)
'102Z....'	
	'ABC%QQR" (either two single quotes or two double quotes must be used)
"A 'LITERAL' STRING"	
'A "LITERAL" STRING'	
'' (indicates the empty string)	"12345678910 (terminating double quote missing)
\JOHN PROGRAMMER\	

Data may also be represented as variables. A variable has a name and a value. The value of a variable may be either numeric or string, and may change dynamically throughout the execution of the program. The name of a variable identifies the variable (the name remains constant throughout program execution). Variable names consist of one alphabetic character followed by zero or more letters, numerals, periods, or dollar signs. A variable name may be from 1 to 64 characters long. A number of valid and invalid variable names:

Valid Variable Name	Invalid Variable Name
A5	ABC 123 (no space allowed)
ABCDEFGHI	
QUANTITY.ON.HAND	5AB (must begin with letter)
R$$$$P$	
J1B2Z	Z.,$ (comma not allowed)
INTEGER	A-B ("-" not allowed)
THIS.IS.A.NAME	

For example:

```
X QUANTITY DATA.LENGTH B$..
```

The variable X, for example, may be assigned the value 100 at the start of a program, and may then later be assigned the value "THIS IS A STRING". Note that BASIC keywords (words that define BASIC statements and functions) may not be used as variable names. BASIC keywords are listed in Tables 1-1 and 1-2, Section 1.1.

2.2 ASSIGNING VALUES TO VARIABLES

The Simple Assignment statement is used to assign a value to a variable. The CLEAR statement is used to initialize all variables to a value of zero.

2.2.1 SIMPLE ASSIGNMENT STATEMENT

The general form of the Simple Assignment statement is:

```
variable = expression
```

The resulting value of the expression becomes the current value of the variable on the left side of the equality sign. The expression may be any legal BASIC expression. For example:

```
ABC = 500
X2  = (ABC+100)/2
```

The first statement will assign the value of 500 to the variable ABC. The second statement will assign the value 300 to the variable X2 (i.e., X2 = (ABC+100)/2 = (500+100)/2 = 600/2 = 300). String values may also be assigned to variables. For example:

```
VALUE = "THIS IS A STRING"
SUB = VALUE [6,2]
```

The first statement above assigns the string "THIS IS A STRING" to variable VALUE. The second statement assigns the string "IS" to variable SUB (it assigns to SUB the 2 character substring starting at character position 6 of VALUE).

2.2.2 CLEAR STATEMENT

The CLEAR statement initializes all possible variables to zero (it assigns the value 0 to all variables). The CLEAR statement may be used in the beginning of the program to initialize all variables to zero, or may be used anywhere within the program for reinitialization purposes.

The general form of the CLEAR statement:

```
CLEAR
```

Examples of Assignment and CLEAR:

Correct Use	Explanation
X=5	Assigns 5 to X.
X=X+1	Increments X by 1.
ST="STRING"	Assigns the character string "STRING" to ST.
ST1=ST[3,1]	Assigns substring "R" to ST1.
TABLE (I,J)=A(3)	Assigns matrix statement from vector element.
A=B=0	Assigns 1 to A if "B=0" is true.* Assigns 0 to A if "B=0" is false.
CLEAR	Assigns the value 0 to all possible variables.

Incorrect Use	Explanation
A=	Expression is missing.
=10+Z	Variable is missing.
A==B	Illegal format.
CLEAR A	A variable is not allowed with CLEAR statement.

*A relational expression evaluates to 1 if the relation is true and to 0 if the relation is false. See Section 3.3, Relational Expressions.

2.2.3 EQUATE STATEMENT

The EQUATE statement allows one variable to be defined as the equivalent of another variable. The general form of the EQUATE statement is:

EQUATE or EQU variable TO equate-variable {, variable TO equate-variable...}

The variable must be a simple variable. The equate-variable may be a literal number, string, character, array element or CHAR function. Note that the CHAR function is the only function allowed in an EQUATE statement.

The EQUATE statement must appear before the first reference to the equate-variable.

The EQUATE statement differs from the ASSIGNMENT Statement (where a variable is assigned a value via an equal sign) in that there is no storage location generated for the variable. The advantage this offers is that the value is compiled directly into the object-code item at compile time and does not need to be reassigned every time the program is executed.

The EQUATE statement is therefore particularly useful under the following two conditions:

1. Where a constant is used frequently within a program, and therefore the program would read more clearly if the constant were given a symbolic name. In the fourth example on the next page, "AM" is the commonly used symbol for "attribute mark", one of the standard data delimiters.

2. Where a MATREAD statement is used to read in a entire item from a file and disperse it into a dimensioned array. In this case, the EQUATE statement may be used to give symbolic names to the individual array elements, which makes the program more meaningful. For example:

```
DIM ITEM(20)

EQUATE BIRTHDATE TO ITEM(1), SOC.SEC.NO.  TO ITEM(2)

EQUATE SALARY TO ITEM(3)
```

In this case, the variables BIRTHDATE, SOC.SEC.NO. and SALARY are rendered equivalent to the first three elements of the array ITEM. These meaningful variables are then used in the remainder of the program.

Examples of the use of EQUATE:

Correct Use	Explanation
EQUATE X TO Y	Variable X and variable Y may be used interchangeably within the program.
EQUATE PI to 3.1416	Variable PI is compiled as the value 3.1416 at compile time.
EQUATE STARS TO "*****"	Variable STARS is compiled as the value of five asterisks at compile time.
EQUATE AM TO CHAR(254)	Variable AM is equivalent to the ASCII character generated by the CHAR function.
EQUATE PART TO ITEM(3)	Variable PART is equivalent to element 3 of array ITEM.

Incorrect Use	Explanation
EQUATE 2.7182 TO E	Variable must appear first.
EQUATE PRICE(9) TO X	Only simple variables (not array elements) may appear after the word EQUATE.

2.3 MULTIPLE DATA REPRESENTATION

Multiple valued variables are called arrays. Before arrays may be used in a BASIC program they must be dimensioned via a DIM statement.

2.3.1 ARRAYS

A variable with more than one value associated with it is called an array. Each value is called an element of the array and the elements are ordered.

```
          -------
          |  3  |---- The first element of A has value 3
          -------
          |  8  |---- The second element of A has value 8
Array A:  -------
          |-20.3|---- The third element of A has value -20.3
          -------
          | ABC |---- The fourth element of A has string value "ABC"
          -------
```

The above example illustrates a one-dimensional array (called a vector). A two-dimensional array (called a matrix) is characterized by having rows and columns. For example:

```
                  COL.1   COL.2   COL.3   COL.4
                 ------------------------------
           Row 1 |  3  | XYZ  |    A  | -8.2 |
                 |-----------------------------
Array Z:   Row 2 |  8  | 3.1  |  500  | .333 |
                 |-----------------------------
           Row 3 |  2  |  -5  | Q123  |  84  |
                 ------------------------------
```

An array element may be accessed by specifying its position in the array. This position is like an offset from the beginning of the array. When specifying an element, the user must have one offset or subscript for each dimension of the array. For example:

```
          --------
          |  -7  |--------- Element B(1)
          |------|
Array B:  |  23  |--------- Element B(2)
          |------|
          |XYZABC|--------- Element B(3)
          --------
```

In this example, element B(1) has a value of -7, while element B(3) has a string value of "XYZABC". For a two-dimensional array (matrix) the first subscript specifies the row, while the second specifies the column. In array Z above, element Z(1,1) has a value of 3, while element Z(2,3) has a value of 500. Before an array may be used in a BASIC program, the maximum dimension(s) of the array must be specified to set aside the correct amount of space for storage. This is done via a DIM statement, which is discussed in the next section.

2.3.2 DIM STATEMENT

Before an array may be used in a BASIC program, it must be dimensioned using a DIM statement. For this reason, DIM statements are usually placed at the beginning of the program. (Arrays need only be dimensioned once throughout the entire program.) The DIM statement has the general form:

```
DIM variable(dimensions){,variable(dimensions)}...
```

The dimensions of any array are specified as constant whole numbers, separated by commas. A dimension of 1 is not allowed. Several arrays may be dimensioned via a single DIM statement. The following statement, for example, declares array A1 as a 10 by 5 matrix and declares array X as a 50 element vector:

```
DIM A1(10,5), X(50)
```

Examples of the use of DIM:

Correct Use	Explanation
DIM MATRIX(10,12)	Specifies 10 by 12 matrix named MATRIX.
DIM Q(10),R(10),S(10)	Specifies three vectors named Q, R, and S, each to contain 10 elements.
DIM M1(50,10),X(2)	Specifies 50 by 10 matrix named M1, and two-element vector named X.

Incorrect Use	Explanation
DIM VECTOR	Dimension subscript is missing.
DIM MATRIX(10 10)	Comma is missing between dimension subscripts
DIM X(10) Y(10,50)	Comma is missing between the X and Y array declarations.
DIM X(1)	Dimensions of 1 are illegal.

2.4 ASSIGNING VALUES TO ARRAYS

MAT Assigment and MAT Copy statements are used to assign values to each element in the array.

2.4.1 MAT ASSIGNMENT STATEMENT

The MAT (matrix) Assignment statement is similar to the Simple Assignment statement. It assigns a single value to all elements in an array. The general form of the MAT Assignment statement is:

```
MAT variable = expression
```

The resulting value of the expression (which may be any legal expression) is assigned to each element of the array. The array that is being assigned is specified by the "variable" parameter. Note that the specified array must have been previously dimensioned via a DIM statement. The following statement assigns the current value of X+Y-3 to each element of array A:

```
MAT A = X+Y-3
```

2.4.2 MAT COPY STATEMENT

The MAT Copy statement copies one array to another. The general form of the MAT Copy statement is:

```
MAT variable = MAT variable
```

The first element of the array on the right becomes the first element of the array on the left, the second element on the right becomes the second element on the left, etc. Each variable name must be previously dimensioned, and the number of elements in the two arrays must match; if not, an error message occurs.

Arrays are copied in row major order (which means that the rows are filled first, therefore, the second subscript (column) will vary first).

Program Code	Resulting Array Values
DIM X(5,2), Y(10)	X(1,1) = Y(1) = 1
FOR I=1 TO 10	X(1,2) = Y(2) = 2
Y(I)=I	X(2,1) = Y(3) = 3
NEXT I	.
MAT X = MAT Y	.
	.
	X(5,2) = Y(10) = 10

The above program dimensions two arrays that each have ten elements (5x2=10), initializes array Y elements to the numbers 1 through 10, and then copies array Y to array X, giving the array elements the indicated values.

Examples of MAT Assignment and MAT Copy:

Correct Use	Explanation
`MAT TABLE=1`	Assigns a value of 1 to each element of array TABLE.
`MAT XYZ=A+B/C`	Assigns the expression value to each element of array XYZ.

`DIM A(20), B(20)`
`  .`
`  .`
`MAT A = MAT B`
	Dimensions two vectors of equal length, and assigns to elements of A, the values of corresponding elements of B.

`DIM TAB1 (10,10), TAB2(50,2)`
`  .`
`  .`
`MAT TAB1 = MAT TAB2`
	Dimensions two arrays of the same number of elements (10x10=50x2), and copies TAB2 values to TAB1 in row major order.

Incorrect Use	Explanation
`MAT=45/Q`	Variable is missing after MAT.
`DIM A(2,2), B(2,2)` `MAT A = B`	Word "MAT" is missing after equality sign.
`DIM AR(3,6), SAVE(20)` `MAT AR = MAT SAVE`	The two arrays are not dimensioned to the same size and thus cannot be made identical.

2.5 SELECTING NUMERIC PRECISION: PRECISION DECLARATION

The PRECISION declaration allows the user to select the degree of precision to which all values will be calculated within a given program.

The default precision value is 4. This means that all values are stored in an internal form with 4 fractional places, and all computations are performed to this degree of precision. However, you may specify the number of fractional digits you desire within the range of 0-4 by using a PRECISION declaration.

The general form of the PRECISION declaration is as follows:

```
PRECISION n
```

where n is a number from 0-4.

Only one PRECISION declaration is allowed in a program. If more than one is encountered, a warning message is printed and the declaration is ignored.

Where external subroutines are used, the mainline program and all external subroutines must have the same PRECISION. If the precision is different between the calling program and the subroutine, a warning message will be printed.

Note that changing the precision changes the acceptable form of a number; a number is defined as having a maximum of "n" fractional digits, where "n" is the precision value. Thus, the value:

```
1234.567
```

is a legal number if the precision is 3 or 4, but is not a legal number if the precision is 0, 1 or 2.

A precision of zero means that all values will be treated as integers.

Examples of the use of PRECISION:

Correct Use	Explanation
PRECISION 0 A = 3 B = A/2	All numeric values in the program will be treated as integers. The value returned for B will be 1, not 1.5.
PRECISION 1	All numeric values in the program will be calculated to one fractional digit.
PRECISION 2	All numeric values in the program will be calculated to two fractional digits.
PRECISION 3	All numeric values in the program will be calculated to three fractional digits.

Incorrect Use	Explanation
PRECISION 5 PRECISION -2	PRECISION must be set within the range of 0-4.
PRECISION 2 A = B + C PRECISION 3	PRECISION may be set only once within a given program. Otherwise, a warning message is issued and the second PRECISION declaration is ignored.
PRECISION 2 CALL SUBA SUBROUTINE SUBA PRECISION 3	PRECISION must be the same for the mainline program and any subroutine it calls. Otherwise, a warning message is issued and the second PRECISION declaration is ignored.
PRECISION 2 A = 12.247	Variable A is assigned with more fractional digits than allowed by the PRECISION declaration.

forming expressions 3

3.1 ARITHMETIC EXPRESSIONS

Expressions are formed by combining operators with variables, constants, or BASIC Intrinsic Functions. Arithmetic expressions are formed by using arithmetic operators.

When an expression is encountered as part of a BASIC program statement, it is evaluated by performing the operations specified by each of the operators on the adjacent operands (i.e., the adjacent constants, identifiers, or intrinsic functions). Arithmetic expressions are formed by using the following arithmetic operators:

Operator Symbol	Operation	Precedence
+	unary plus	1 (high)
-	unary minus	1
*	multiplication	2
/	division	2
+	addition	3
-	subtraction	3 (low)

The simplest arithmetic expression is a single numeric constant, variable, or intrinsic function. A simple arithmetic expression may combine two operands using an arithmetic operator. More complicated arithmetic expressions are formed by combining simple expressions using arithmetic operators.

When more than one operator appears in an expression, certain rules are followed to determine which operation is to be performed first. Each operator has a precedence rating. In any given expression, the highest precedence operation will be performed first. Note that the + and - signs both perform two different operations. The first, which has the highest precedence, establishes the positive or negative value of an expression. Because this type of + and - will only have an expression on one side, this is called the unary plus and minus. The + and - which perform ordinary addition and substration (the lowest precedence) will have expressions on both sides. If there are two or more operators with the same precedence (or an operator appears more than once) the leftmost operation is performed first.

For example, consider this expression: -R/A+B*C. The unary minus is evaluated first (-R = Result 1). The expression then becomes:

Result 1 / A+B*C

The division and multiplication operators have the same precedence; since the division operator is leftmost it is evaluated next (Result 1 / A = Result 2). The expression then becomes:

Result 2 + B*C

The multiplication operation is performed next (B*C = Result 3). The Result 2 + Result 3 = Final Result.

Placing some figures in the above expression, for example, -50/5+3*2, illustrates that the expression evaluates to -4.

Any subexpression may be enclosed in parentheses. Within the parentheses, the rules of precedence apply, therefore, the parenthesized subexpression as a whole has highest precedence and is evaluated first. For example: (10+2)*(3-1) = 12*2 = 24. Parentheses may be used anywhere to clarify the order of evaluation even if they do not change the order.

If a string expression containing only numeric characters is used in an arithmetic expression, it is considered as a decimal number. For example, 123 + "456" evaluates to 579. Note that at this time, "." is considered a decimal point and has the value 0. In a future release of the system, "." will be nonnumeric.

If a string value containing non-numeric characters is used in an arithmetic expression, a warning message will be printed (refer to Appendix C, BASIC RUN-TIME ERROR MESSAGES) and zero will be assumed for the string value.

For example, 123 + "ABC" evaluates to 123.

Examples of the use of arithmetic expressions:

Correct Use	Explanation
2+6+8/2+6	Evaluates to 18
12/2*3	Evaluates to 18
12/(2*3)	Evaluates to 2
A+75/25	Evaluates to 3 plus the current value of variable A
-5+2	Evaluates to -3
-(5+2)	Evaluates to -7
8*(-2)	Evaluates to -16
5 * "3"	Evaluates to 15
Incorrect Use	**Explanation**
A+/B	Operand is missing between "+" and "/".
(Q1+5)(8+Z)	An operator is missing between the two parenthesized subexpressions.
10-VAL+	Trailing operand is missing.
66*"ABC"	Illegal expression (evaluates to 0).

3.2 STRING EXPRESSIONS

A string expression may be any of the following: a string constant, a variable with a string value, a substring, or a concatenation of string expressions. The general form of string expressions is:

```
variable [expression,expression]
```

A substring is a set of characters which makes up part of a whole string. For example, "S.", "123", and "ST." are substrings of the string "1234 S. MAIN ST." Substrings are specified by a starting character position and a substring length, separated by a comma and enclosed in square brackets. For example, if the current value of variable S is the string "ABCDEFG", then the current value of S[3,2] is the substring "CD" (the two character substring starting at character position 3 of string S). Furthermore, the value of S[1,1] would be "A", and the value of S[2,6] would be "BCDEFG".

If the "starting character" specification is past the end of the string value, then an empty substring value is selected (if A has a value of 'XYZ', then A[4,1] will have a value of ''). If the "starting character" specification is negative or zero, then the first character is assumed (if X has a value of 'JOHN', then X[-5,1] will have a value of 'J').

If the "substring length" specification exceeds the remaining number of characters in the string, then the remaining string is selected (if B has a value of '123ABC', the B[5,10] will have a value of 'BC'). If the "substring length" specification is negative or zero, then an empty substring is selected (B[5,-2] and B[5,0] both have a value of '').

Concatenation operations may be performed on strings. Concatenation is specified by a colon (:) or CAT operator. The concatenation of two strings (or substrings) is the addition of the characters of the second operand into the end of the first. For example:

```
"AN EXAMPLE OF " CAT "CONCATENATION"
```

evaulates to:

```
"AN EXAMPLE OF CONCATENATION"
```

The precedence of the concatenation operator is higher than any of the arithmetic operators. So if the concatenation operator appears in the same expression with an arithmetic operator, the concatenation operation will be performed first. Multiple concatenation operations are performed from left to right. Parenthesized subexpressions are evaluated first. The concatenation operator considers both its operands to be string values; for example, the expression 56:"ABC" evaluates to "56ABC".

Examples of STRING expressions:

NOTE: For the following examples, assume that the current value of A is "ABC123", and the current value of variable Z is "EXAMPLE".

Correct Use	Explanation
Z[1,4]	Evaluates to "EXAM".
A : Z[1,1]	Evaluates to "ABC123E".
Z[1,1] CAT A[4,3]	Evaluates to "E123".
5*2:0	2:0 is evaluated first and results in the string "20" (the concatenation operator assumes both operands are strings). 5*"20" is then evaluated and results in 100 (the * operator assumes both operands are numeric). Final result is 100.
A[6,1]+5	Evaluates to 8.
Z CAT A : Z	Evaluates to "EXAMPLEABC123EXAMPLE".
Z CAT " ONE"	Evaluates to "EXAMPLE ONE".

Incorrect Use	Explanation
"AB" CAT "CD" + 5	"AB" CAT "CD" evaluates to "ABCD"; but "ABCD" + 5 is illegal.
43 * 15 CAT "J"	15 CAT "J" evaluates to "15J"; but 43 * "15J" is illegal.

3.3 RELATIONAL EXPRESSIONS

Relational expressions are the result of applying a relational operator to a pair of arithmetic or string expressions. The relational operators are listed below. A relational operation evaluates to 1 if the relation is true, and evaluates to 0 if the relation is false. Relational operators have lower precedence than all arithmetic and string operators; therefore, relational operators are only evaluated after all arithmetic and string operations have been evaluated.

Operator Symbols	Operation
< or LT	Less than
> or GT	Greater than
<= or LE or =<	Less than or equal to
>= or GE or =>	Greater than or equal to
= or EQ	Equal to
# or <> or NE	Not equal to
MATCH or MATCHES	Pattern matching (this relational operator is discussed in Section 3.3.1).

For clarity, relational expressions may be divided into two types: arithmetic relations and string relations. An arithmetic relation is a pair of arithmetic expressions separated by any one of the relational operators.

3 < 4	(3 is less than 4)=(TRUE)=1
3 = 4.0	(3 is equal to 4)=(FALSE)=0*
3 GT 3	(3 is greater than 3)=(FALSE)=0
3 >= 3	(3 is greater than or equal to 3)=(TRUE)=1
5+1 > 4/2	(5 plus 1 is greater than 4 divided by 2)=(TRUE)=1

A string relation is a pair of string expressions separated by any one of the relational operators. A string expression containing only numeric characters is treated as an arithmetic expression in a relational expression (see Section 3.3.1, Pattern Matching, for a description of string relations with these type of strings. A string relation may also be a string expression and an arithmetic expression separated by a relational operator. If a relational operator encounters one numeric operand and one string operand that does not contain only numeric characters, it treats both operands as strings.

To resolve a string relation, character pairs (one from each string) are compared one at a time from leftmost characters to rightmost. If no unequal character pairs are found, the strings are considered to be 'equal'. If an unequal pair of characters are found, the characters are ranked according to their numeric ASCII code equivalents (refer to Appendix A, ASCII CODES). The string contributing the higher numeric ASCII code equivalent is considered to be "greater" than the other string.

*At an earlier time, "." was evaluated as 0 and caused 0=. to evaluate TRUE. In release 2.1, "." is nonnumeric and 0="." evaluates FALSE.

Consider the following relation:

"AAB" > "AAA"

This relation evaluates to 1 (TRUE) since the ASCII equivalent of B (X'42') is greater than the ASCII equivalent of A (X'41').

If the two strings are not the same length, but the shorter string is otherwise identical to the beginning of the longer string, then the longer string is considered "greater" than the shorter string. The following relation, for example, is TRUE and evaluates to 1:

"STRINGS" GT "STRING"

Examples of the use of relational operators:

Correct Use	Explanation
4 < 5	Evaluates to 1 (TRUE).
"D" EQ "A"	Evaluates to 0 (FALSE).
"D" > "A"	ASCII equivalent of D(X'44') is greater than ASCII equivalent of A(X'41'), so expression evaluates to 1.
"Q" LT 5	ASCII equivalent of Q(X'51') is not less than ASCII equivalent of 5(X'35'), so expression evaluates to 0.
6+5 = 11	Evaluates to 1.
Q EQ 5	Evaluates to 1 if current value of variable Q is 5; evaluates to 0 otherwise.
"ABC" GE "ABB"	Evaluates to 1 (C is "greater" than B)
"XXX" LE "XX"	Evaluates to 0.

Incorrect Use	Explanation
"BB" ET "AB"	ET is not a relational operator
5 EQ GT Z	EQ and GT may not appear adjacent to each other.
6+5>	Second operand is missing.

3.3.1 PATTERN MATCHING

BASIC pattern matching is a relational expression which allows the comparison of a string value to a predefined pattern. Pattern matching is specified by the MATCH or MATCHES relational operator. The general form of the pattern matching relation is:

```
expression MATCH "pattern"

           or

expression MATCHES "pattern"
```

The MATCH or MATCHES relational operator compares the string value of the expression to the predefined pattern (which is also a string value) and evaluates to 1 (TRUE) or 0 (FALSE). The pattern must be enclosed in quotes and may consist of any combination of the following:

- An integer number followed by the letter N (which tests for that number of numeric characters).
- An integer number followed by the letter A (which tests for that number of alphabetic characters).
- An integer number followed by the letter X (which tests for that number of any characters).
- A literal string enclosed in quotes (which tests for that literal string of characters).

Consider the following expression:

```
DATA MATCHES "4N"
```

This relation evaluates to 1 if the current string value of variable DATA consists of four numeric characters.

If the integer number used in the pattern is 0, then the relation will evaluate to TRUE only if all the characters in the string conform with the "specification letter" (with N, A, or X). For example:

```
X MATCH "0A"
```

This relation evaluates to 1 if the current string value of variable X consists only of alphabetic characters. As a further example, consider:

```
A MATCHES "1A4N"
```

This relation evaluates to 1 if the current string value of variable A consists of an alphabetic character followed by four numeric characters.

Examples of the correct and incorrect use of pattern matching:

Correct Use	Explanation
Z MATCHES ´9N´	Evaluates to 1 if current string value of variable Z consists of 9 numeric characters; evaluates to 0 otherwise.
Q MATCHES "ON"	Evaluation to 1 if current value of Q is any unsigned integer; evaluates to 0 otherwise.
B MATCH ´3N"-"2N"-"4N´	Evaluates to 1 if current value of 8 is any social security number; evaluates to 0 otherwise.
B="4N1A2N" C MATCHES B	Evaluates to 1 if current string value of C consists of four numeric characters followed by one alphabetic character followed by two numeric characters; evaluates to 0 otherwise.
A MATCHES "ON´ . ´ON"	Evaluates to 1 if current value of A is any number containing a decimal point; evaluates to 0 otherwise.
"ABC" MATCHES "#N"	Evaluates to 0.
"XYZ" MATCHES "A"	Evaluates to 1.
"XYZ1" MATCH "4X"	Evaluates to 1.
X MATCHES ´´	Evaluates to 1 if current string value of X is the empty string; evaluates to 0 otherwise.
0 MATCH ´.´	Evaluates to 0.

Incorrect Use	Explanation
DATA MATCH "3M"	"3M" is not a legal pattern specification.
Z MATCHES "X"	An integer number must precede X.
Q MATCH "3AN"	An integer number is missing between A and N.

3.4 LOGICAL EXPRESSIONS

Logical expressions (also called Boolean expressions) are the result of applying logical (Boolean) operators to relational or arithmetic expressions. The logical operators are:

Operator Symbol	Operation
AND	Logical AND operation
OR	Logical OR operation

Logical operators operate on the true or false results of relational or arithmetic expressions. (Relational expressions are considered FALSE when equal to zero, and are considered TRUE when equal to one; arithmetic expressions are considered FALSE when equal to zero, and are considered TRUE when not equal to zero.) Logical operators have the lowest precedence and are only evaluated after all other operations have been evaluated. If two or more logical operators appear in an expression, the leftmost is performed first. Logical operators act on their associated operands as follows:

a OR b — is TRUE (evaluates to 1) if a is TRUE or b is TRUE or both are TRUE;
is FALSE (evaluates to 0) only when a and b are both FALSE.

a AND b — is TRUE (evaluates to 1) only if both a and b are TRUE;
is FALSE (evaluates to 0) if a is FALSE or b is FALSE or both are FALSE.

Consider the following logical expression:

```
A*2-5>B AND 7>J
```

The multiplication operation has highest precedence, so it is evaluated first (A*2 = Result 1). The expression then becomes:

```
Result 1 - 5>B AND 7>J
```

The subtraction operation is next (Result 1 - 5 = Result 2). The expression then becomes:

```
Result 2 > B AND 7>J
```

The relational operators are of equal precedence, so the leftmost is evaluated first (Result 2 > B = Result 3, where Result 3 has a value of 1 indicating TRUE, or a value of 0 indicating FALSE). The expression then becomes:

```
Result 3 AND 7>J
```

The remaining relation operation is then performed (7>J = Result 4, where Result 4 equals 1 or 0). The final expression therefore becomes:

Result 3 AND Result 4

which is evaluated as TRUE (1) if both Result 3 and Result 4 are TRUE, and is evaluated as FALSE (0) otherwise.

The NOT function may be used in logical expressions to negate (invert) the expression or subexpression (refer to the description of the NOT Intrinsic Function). Examples of the use of logical expressions:

Correct Use	Explanation
1 AND A	Evaluates to 1 if current value of variable A is non-zero; evaluates to 0 if current value of A is 0.
8-2*4 OR Q5-3	Evaluates to 1 if current value of Q5-3 is non-zero; evaluates to 0 if current value of Q5-3 is 0.
A>5 OR A<0	Evaluates to 1 if the current value of variable A is greater than 5 or is negative; evaluates to 0 otherwise.
1 AND (0 OR 1)	Evaluates to 1.
J EQ 7 AND I EQ 5*2	Evaluates to 1 if the current value of variable J is 7 and the current value of variable I is 10; evaluates to 0 otherwise.
"XYZ1" MATCH "4X" AND X	Evaluates to 1 if the current value of variable X is non-zero; evalutes to 0 if current value of X is 0.
X1 AND X2 AND X3	Evaluates to 1 if the current value of each variable (X1, X2, and X3) is non-zero; evaluates to 0 if the current value of either or all variables is 0.

Incorrect Use	Explanation
B=5 AND	Second operand is missing.
A AND OR X<9	Operand is missing between "AND" and "OR".
A ORB	A blank space is missing between "OR" and "B".

program control and operation 4

4.1 UNCONDITIONAL BRANCHING

4.1.1 GOTO STATEMENT

The GOTO statement unconditionally transfers program control to any statement within the BASIC program. The general form of the GOTO statement is:

```
GOTO statement-label    or    GO statement-label
```

Execution of the GOTO or GO statement causes program control to transfer to the statement which begins with the specified numeric statement-label. If a statement does not exist with the specified statement-label, an error message will be printed at compile time (see Appendix B, COMPILER ERROR MESSAGES). An illustration of correct use of the GOTO statement is given below. (Note that statements to test and end repetition are assumed in the body of the program.)

```
+====> 100 A=D
|           .
|           .
|           REM BRANCH TO STATEMENT 500
|       200 GOTO 500 ==+
|           .          |
|           .          |
|  +=================+
|  |        .
|  +=> 500 A=B+C
|           D=100
|           .
|           .
|           REM REPEAT PROGRAM
|           GOTO 100 ==+
+=====================+
```

The flow of program control is illustrated by the arrows. Note that control may be transferred to statements following the GOTO statement, as well as to statements preceding the GOTO statement. An example of incorrect use of the GOTO statement:

```
          100 A=0                    Explanation
              .
              .              Program will permanently "hang", keep
      ==> 200 GOTO 200 ==    reexecuting this statement.
      +================+
```

4.2 COMPUTED BRANCHING

4.2.1 ON GOTO STATEMENT

The ON GOTO statement transfers control to one of several statement-labels selected by the current value of an index expression. The general form of the ON GOTO statement is:

```
ON expression GOTO statement-label, statement-label,...
```

Upon execution of the ON GOTO statement, program control is transferred to the statement which begins with the numeric statement-label selected by the expression. Statement-labels in the list are numbered 1, 2, 3,.... In executing the ON GOTO statement, the expression is evaluated and then the result of the expression is truncated to an integer value. Consider the following example:

```
ON I GOTO 50, 100, 150
.
.
. 50  .
.
.
. 100 .
.
. 150 .
.
.
```

(Note that the labels in the label list may precede or follow the ON GOTO statement.) If the current value of variable I=1, control transfers to the first statement-label, (i.e., the statement with label 50). If I=3, control transfers to the third statement-label, (i.e., statement 150).

If the value of the expression evaluates to less than one or greater than the number of statement-labels, no action is taken, which means that the statement immediately following the ON GOTO will be executed next.

Examples of the usage of ON GOTO:

Correct Use	Explanation
`ON M+N GOTO 40, 61, 5, 7`	Transfer control to statement 40, 61, 5, or 7 depending on the value of M+N being 1, 2, 3, or 4 respectively.
`ON C GOTO 25, 25, 20`	Transfer control to statement 25 if C = 1 or 2, to statement 20 if C = 3.

`IF A GE 1 AND A LE 3 THEN`
`  ON A GOTO 110, 120, 130`
`END`
	The IF statement assures that A is in range for the computed GOTO statement.

Incorrect Use	Explanation
`ON A 100, 200`	"GOTO" missing.
`ON MAX GOTO M(10)`	Invalid statement-label (must be numeric).
`ON M+N=1 GOTO 40, 61, 5, 7`	Index should be arithmetic, not logical, quantity. This statement, if executed, would cause an unconditional jump to statement 40.

4.3 CONDITIONAL BRANCHING

4.3.1 SINGLE-LINE IF STATEMENT

The Single-Line IF statement provides the conditional execution of a sequence of BASIC statements or the conditional execution of one of two sequences of statements, which appear on a single line. The general form of the Single-Line IF statement is:

```
IF expression THEN statement(s) {ELSE statement(s)}
```

If the result of the test condition specified by the expression is true (i.e., non-zero), then the statement or sequence of statements following the THEN is executed. If the result of the expression is false (i.e., zero), then the statement or sequence of statements following the ELSE is executed, unless the ELSE clause is omitted, in which case, control will pass to the next sequential statement following the entire IF statement. The expression may be any legal BASIC expression.

The sequence of statements in the THEN or ELSE clauses may consist of one or more statements on the same line. For example:

```
IF X>1 THEN GOTO 50
```

In this example, control will be transferred to statement 50 if the current value of X is greater than one. Since the ELSE clause is not used here, control will pass to the next statement in the program if X is not greater than one.

If more than one statement is contained in either the THEN or ELSE clause, they must be separated by semicolons. Consider the example:

```
IF ITEM THEN PRINT X;  X=X+1 ELSE PRINT X*5;  GOTO 10
```

If the current value of ITEM is non-zero (i.e., true), then this statement will print the current value of X, add one to the current value of X, and then transfer control to the next sequential instruction in the program. If the value of ITEM is zero (i.e., false), then the value of X*5 will be printed and control will transfer to statement 10.

Any statements may appear in the THEN or ELSE clauses, including additional IF statements. For example:

```
IF X THEN NULL ELSE IF C THEN GOTO 10
```

The THEN clause of an IF statement is optional if the ELSE clause is present. However, one or the other must be present.

Examples of single line IF:

Correct Use	Explanation
IF A="STRING" THEN PRINT "MATCH"	Prints "MATCH" if value of A is the string "STRING".
IF X>5 THEN IF X<9 THEN GOTO 10	Transfers control to statement 10 if X is greater than 5 but less than 9.
IF Q THEN PRINT A ELSE PRINT B; STOP	The value of A is printed if Q is non-zero. If Q=0, then the value of B is printed and the program is terminated.
IF A=B THEN STOP ELSE IF C THEN GOTO 20	Program is terminated if A=B; control is passed to statement 20 if A does not equal B and if C is non-zero.

Incorrect Use	Explanation
IF A#B THEN 50	The word "GOTO" is missing between "THEN" and "50".
IF X>1 THEN ELSE GOTO 10	At least one statement must appear between "THEN" and "ELSE".
IF A = 10 GOTO 20	The word "THEN" is missing between 10 and GOTO.
IF D THEN X=A+D GOTO 3	A semicolon is missing between "X=A+D" and "GOTO 3".

4.3.2 MULTI-LINE IF STATEMENT

The Multi-Line IF statement functions in the same way as the Single-Line IF statement. It provides the conditional execution of a sequence of BASIC statements, or the conditional execution of one of two sequences of statements. The statement sequences, however, may be placed on multiple program lines. The general forms of this statement:

```
FORM 1:     IF expression THEN
            statements
            .
            .
            .
            END ELSE statements

FORM 2      IF expression THEN
            statements
            .
            .
            .
            END ELSE
            statements
            .
            .
            .
            END

FORM 3:     IF expression THEN
            statements ELSE
            .
            .
            .
            END
```

NOTE: In each of the above forms, the ELSE clause is optional.

The Multi-Line IF statement is actually an extension of the Single-Line format. With this format, the statement sequences in the THEN and ELSE clauses may be placed on multiple program lines, which each sequence terminated by an END.

In each of the three forms, the ELSE clause is optional and may be included or omitted as desired. Any statements may appear in the THEN and ELSE clauses.

Examples of Multi-Line IF:

Correct Use

```
IF ABC=ITEM+5 THEN
   PRINT ABC
   STOP
END ELSE PRINT ITEM; GOTO 10
```

The value of ABC is printed and the program terminates if ABC=ITEM+5; otherwise, the value of ITEM is printed and control passes to statement 10.

```
IF VAL THEN
   PRINT MESSAGE
   PRINT VAL
   VAL=100
   END
```

If the value of VAL is non-zero, then the value of MESSAGE is printed, the value of VAL is printed, and VAL is assigned a value of 100; otherwise, control passes to the next statement following END.

```
10 IF S="XX" THEN PRINT "OK" ELSE
   PRINT "NO MATCH"
   PRINT S
   STOP
   END
20 REM REST OF PROGRAM
```

If the value of S is the string "XX", then the message "OK" is printed and control passes to statement 20; otherwise, "NO MATCH" is printed, the value of S is printed, and the program terminates.

```
IF X>1 THEN
   PRINT X
   X=X+1
END ELSE
   PRINT "NOT GREATER"
   GOTO 75
END
```

If X>1 the value of X is printed and then incremented, and control passes to the next statement following the second END: otherwise "NOT GREATER" is printed and control passes to statement 75.

Incorrect Use

```
IF X>=Y THEN B=1
   ELSE B=2
   PRINT Y
   END
```

ELSE must appear at the end of the first line rather than at the beginning of the second.

```
IF Q=1 THEN
   A=A+1
   B=B+1
ELSE A=0; B=0
```

END is missing (i.e., END must precede ELSE in this example).

4.3.3 CASE STATEMENT

The CASE statement provides conditional selection of a sequence of BASIC statements. The CASE statement has the following general form:

```
BEGIN CASE
    CASE expression
    statements
    CASE expression
    statements
    .
END CASE
```

If the logical value of the first expression is true (i.e., non-zero), then the statement or sequence of statements that immediately follows is executed, and control passes to the next sequential statement following the entire CASE statement sequence. If the first expression is false (i.e., zero), then control passes to the next test expression, and so on. Consider the following example:

```
BEGIN CASE
    CASE A  <  5
    PRINT 'A IS LESS THAN 5'
    CASE A  <  10
    PRINT 'A IS GREATER THAN OR EQUAL TO 5 AND LESS THAN 10'
    CASE 1
    PRINT 'A IS GREATER THAN OR EQUAL TO 10'
END CASE
```

If A<5, then the first PRINT statement will be executed. If 5<=A<10, then the second PRINT statement will be executed. Otherwise, the third PRINT statement will be executed. Note that a test expression of 1 means "always true".

WARNING: Programs containing more END CASE statements than BEGIN CASE statements will compile successfully, but will terminate with message "[B15] LINE 'A' ILLEGAL OPCODE: 'C'" when the extra END CASE statement is encountered at run time.

Examples of the of CASE:

Correct Use	Explanation

BEGIN CASE
  CASE Y=B
  Y=Y+1
END CASE
	Increment Y if Y is equal to B. Note that this single-case example is equivalent to the statement IF Y=B THEN Y=Y+1.

BEGIN CASE
  CASE A=0; GOTO 10
  CASE A<0; GOTO 20
  CASE 1; GOTO 30
END CASE
	Program control branches to the statement with label 10 if the value of A is zero; to 20 if A is negative; or to 30 if A is greater than zero.

BEGIN CASE
  CASE ST MATCHES "1A"
  MAT LET=1
  CASE ST MATCHES "1N"
  SGL=1; A.1(I)=ST
  CASE ST MATCHES "2N"
  DBL=1; A.2(J)=ST
  CASE ST MATCHES "3N"
  GOSUB 103
END CASE
	If ST is one letter, "1" is assigned to all LET elements and the entire CASE is ended. If ST is one number, "1" is assigned to SGL, ST is stored at element A.1(I), and the entire case is ended. If ST is two numbers, "1" is assigned to DBL, ST is stored at element A.2(J), and the entire case is ended. If ST is three numbers, subroutine 103 is executed.

Incorrect Use	Explanation
CASE X=Y GOTO 100 END CASE	'BEGIN CASE' statement is missing.

BEGIN CASE
  CASE SS=1, SS=0
  CASE 1; SS=SS-1
	'END CASE' statement is missing.

BEGIN CASE A+B<C A=C/2 END CASE	The case condition, A+B<C, must be a separate statement preceded by its own word 'CASE'.

BEGIN CASE
  CASE N=100
  CASE N>100
END CASE
	Executable statements are missing.

4.4 NO OPERATIONS

4.4.1 NULL STATEMENT

The NULL statement specifies a non-operation and may be used in situations where a BASIC statement is required, but no operation or action is desired. Consider the following example:

```
IF X1 MATCHES "9N" THEN NULL ELSE GOTO 100
```

This statement will cause program control to branch to statement 100 if the current string value of variable X1 does not consist of 9 numeric characters. If the current string value of variable X1 does consist of 9 numeric characters, then no action will be taken and program control will proceed to the next sequential BASIC statement.

The NULL statement may be used anywhere in the BASIC program where a statement is required. Examples of the use of NULL:

Correct Use	Explanation
10 NULL	This statement does not result in any operation or action; however, since it is preceded by a statement label (10) it may be used as a program entry point for GOTO or GOSUB statements elsewhere in the program.
IF A=0 THEN NULL ELSE PRINT "A NON-ZERO" GOSUB 45 STOP END	If the current value of variable A is non-zero, then the sequence of statements following the ELSE will be executed. If A=0, no action is taken and control passes to the next sequential statement following the END.
READ A FROM "ABC" ELSE NULL	File item ABC is read and assigned to variable A. If ABC does not exist, no action is taken (refer to READ statement, Section 7.3.1).

Incorrect Use	Explanation
NULL X1	A parameter is not allowed with the NULL statement.
IF NULL THEN GOTO 5	"NULL" may not be used as an expression.
NULL=45*B	"NULL" may not be used as a variable name.

4.5 PROGRAM LOOPING

4.5.1 FOR AND NEXT STATEMENTS

The FOR and NEXT statements are used to specify the beginning and ending points of a program loop. A loop is a portion of a program written in such a way that it will execute repeatedly until some test condition is met. The general form:

```
FOR variable = expression TO expression {STEP expression}
                    |             |              |
NEXT variable       |             |              |
                 --------    ----------    -----------
                |initial|   |limiting|    |increment|
                |value  |   |value   |    |value    |
                --------    ----------    -----------
```

A FOR and NEXT loop causes execution of a set of statements for successive values of a variable until a limiting value is encountered. Such values are specified by establishing: 1) an initial value for a variable, 2) a limiting value for the variable, and 3) an increment value to be added to the value of the variable at the end of each pass through the loop. When the limit is exceeded, program control proceeds to the body of the program following the loop.

The expression preceding TO specifies the initial value of the variable, the expression following TO gives the limiting value, and the optional expression following STEP gives the increment. If STEP is omitted, the increment value is assumed to be +1. The initial value expression is evaluated only once (when the FOR statement is executed). The other two expressions are evaluated on each iteration of the loop.

The function of the NEXT statement is to return program control to the beginning of the loop after a new value of the variable has been computed.

Note that the variable in the NEXT statement must be the same as the variable in the FOR statement. Gonsider the execution of the following statements:

```
150 FOR J=2 TO 11 STEP 3
160 PRINT J+5
170 NEXT J
```

Statement 150 sets the initial value of J to 2 and specifies that J thereafter will be incremented by 3 each time the loop is performed, until J exceeds the limiting value 11. Statement 160 prints out the current value of the expression J+5. Statement 170 assigns J its next value (i.e., J=2+3=5) and causes program control to return to statement 150. Statement 160 is again executed, and statement 170 again increments J and causes the program to loop back. This process continues with J being incremented by 3 after each pass through the loop.

When J attains the limiting value of 11, statement 160 will again be executed and control will pass to 170. J will again be incremented (i.e., J=11+3=14), and since 14 is greater than the limiting value of 11, the program will "fall through" statements 160 and 170, and control will pass to the next sequential statement following statement 170.

Examples of the use of FOR and NEXT:

Correct Use	Explanation
FOR A=1 TO 2+X-Y . . NEXT A	Limiting value is current value of expression 2+X-Y; increment value is +1.
FOR K=10 TO 1 STEP -1 . . NEXT K	Increment value is -1 (variable K will decrement by a value -1 for each of 10 passes through the loop).
FOR VAR= 0 TO 1 STEP .1 . . NEXT VAR	Increment value is .1 (variable VAR WILL increment by a value of .1 for each of 11 passes through the loop).

Incorrect Use	Explanation
FOR 1 TO 50 STEP 5	Variable is missing.
FOR X=1 STEP 2	Limiting value is missing.
FOR J=5 TO 1 STEP 2	Increment value must be negative.
FOR Y=1 TO 10 STEP -1	Increment value must be positive.

4.5.1.1 WHILE and UNTIL Clauses

The condition clauses WHILE and UNTIL may be used in the FOR statement. The FOR statement may be used in the following extended forms:

```
FOR variable = expression TO expression {STEP
               expression}{WHILE expression}

FOR variable = expression TO expression {STEP
               expression}{UNTIL expression}
```

The extended form of the FOR statement functions in the same way as the basic FOR statement with the following additions.

If the WHILE clause is used, the expression specified in the clause will be evaluated for each iteration of the loop. If it evaluates to false (i.e., zero), then program control will pass to the statement immediately following the accompanying NEXT statement. If it evaluates to true (i.e., non-zero), the loop will reiterate.

If the UNTIL clause is used, the expression specified in the clause will be evaluated for each iteration of the loop. If it evaluates to true (i.e., non-zero), then program control will pass to the statement immediately following the accompanying NEXT statement. If it evaluates to false (i.e., zero), the loop will reiterate.

The following FOR and NEXT loop, for example, will execute until I=10 or until the statements within the loop cause variable A to exceed the value 100:

```
FOR I=1 TO 10 STEP .5 UNTIL A>100
 .
 .
 .
NEXT I
```

4.5.1.2 Nesting

FOR and NEXT loops may be contained within the range of other FOR and NEXT loops. These loops are called nested loops. An example of a nested loop;

```
FOR I=1 TO 10
  FOR J=1 TO 10
  PRINT B (I,J)
  NEXT J
NEXT I
```

The above statements illustrate a two-level nested loop. The inner loop will be executed ten times for each of ten passes through the outer loop, i.e., the statement PRINT B(I,J) will be executed 100 times, causing matrix B to be printed in the following order: B(1,1), B(1,2), B(1,3),..., B(1,10), B(2,1), B(2,2),..., B(10,10).

Loops may be nested any number of levels. However, a nested loop must be completely contained within the range of the outer loop (i.e., the ranges of the loops may not cross).

Examples of extended use of FOR and NEXT:

Correct Use	Explanation

`ST="X"`
`FOR B=1 TO 10 UNTIL ST="XXXXX"`
`ST=ST CAT "X"`
`NEXT B`
	Loop will execute 4 times (i.e., an "X" is added to the string value of variable ST until the string equals "XXXXX").

`A=20`
`FOR J=1 TO 10 WHILE A<25`
`A=A+1`
`PRINT J,A`
`NEXT J`
	Loop will execute 5 times (i.e., variable A reaches 25 before variable J reaches 10).

`A=0`
`FOR J=1 TO 10 WHILE A<25`
`A=A+1`
`PRINT J,A`
`NEXT J`
	Loop will execute 10 times (i.e., variable J reaches 10 before variable A reaches 25).

Incorrect Use	Explanation
`FOR X=6 WHILE B=0`	"TO expression" is missing.
`FOR B=1 TO 7 UNTIL`	Expression is missing after "UNTIL".
`FOR I=0 TO 3 UNTIL X STEP .5`	"STEP .5" must appear before "UNTIL X".

4.5.2 LOOP STATEMENTS

Program loops may also be constructed by using the LOOP statement. The LOOP statement may be used in either of the following two general forms:

```
LOOP {statements} WHILE expression DO {statements} REPEAT

LOOP {statements} UNTIL expression DO {statements} REPEAT
```

Execution of a LOOP statement proceeds as follows. First the statements (if any) following "LOOP" will be executed. Then the expression is evaluated. One of the following is then performed depending upon the form used:

- When the "WHILE" form is used, the statements following "DO" (if any) will be executed and program control will loop back to the beginning of the loop if the WHILE expression evaluates to TRUE (non-zero). Otherwise, program control will proceed with the next sequential statement following "REPEAT" (control passes out of the loop if the expression evaluates to FALSE, i.e., zero).

- When the "UNTIL" form is used, the statements following "DO" (if any) will be executed and program control will loop back to the beginning of the loop if the UNTIL expression evaluates to FALSE (zero). Otherwise, program control will proceed with the next sequential statement following "REPEAT" (control passes out of the loop if the expression evaluates to TRUE, i.e., non-zero).

Statements used within the LOOP statement may be placed on one line separated by semicolons, or may be placed on multiple lines. Consider the following example:

```
LOOP UNTIL A=4 DO A=A+1;  PRINT A REPEAT
```

Assuming that the value of variable A is 0 when the LOOP statement is first executed, this statement will print the sequential values of A from 1 through 4 (i.e., the loop will execute 4 times). As a further example, consider the statement:

```
LOOP X=X-10 WHILE X>40 DO PRINT X REPEAT
```

Assuming, for example, that the value of variable X is 100 when the above LOOP statement is first executed, this statement will print the values of X from 90 down through 50 in increments of -10 (i.e., the loop will execute 5 times).

Examples of the use of LOOP:

Correct Use	Explanation
J=0 LOOP PRINT J J=J+1 WHILE J<4 DO REPEAT	Loop will execute 4 times (i.e., sequential values of variable J from 0 through 3 will be printed).
Q=6 LOOP Q=Q-1 WHILE Q DO PRINT Q REPEAT	Loop will execute 5 times (i.e., values of variable Q will be printed in the following order: 5, 4, 3, 2, and 1).
Q=6 LOOP PRINT Q WHILE Q DO Q=Q-1 REPEAT	Loop will execute 7 times (i.e., values of variable Q will be printed in the following order: 6, 5, 4, 3, 2, 1, and 0).
B=1 LOOP UNTIL B=6 DO B=B+1 PRINT B REPEAT	Loop will execute 5 times (i.e., sequential values of variable B from 2 through 6 will be printed).

Incorrect Use	Explanation
LOOP UNTIL B=5 DO B=B+1	"REPEAT" is missing.
LOOP DO K=K*K;PRINT K REPEAT	"UNTIL" or "WHILE" (followed by an expression) is missing.
A=5 LOOP WHILE A>0 DO PRINT A REPEAT	Loop will execute indefinitely.

4.6 PROGRAM TERMINATION

4.6.1 END, STOP AND ABORT STATEMENTS

When the END statement is the last statement of the BASIC program, it designates the physical end of the program. The STOP and ABORT statements, which may appear anywhere in the program, designate a logical termination of the program.

The END statement may appear as the very last statement in the BASIC program. It is used to specify the physical end of the sequence of statements comprising the program. The general form of the END statement is:

```
END
```

The END statement is also used to designate the physical end of alternative sequences of statements within the IF statement and within some of the BASIC I/O statements.

The STOP and ABORT statements may be placed anywhere within the BASIC program to indicate the end of one of several alternative paths of logic. Upon the execution of a STOP or ABORT statement, the BASIC program will terminate. In addition, the ABORT statement will terminate execution of any PROC which may be active.

The STOP and ABORT statements may optionally be followed by an error message name, and error message parameters separated by commas. The error message name is a reference to an item in the ERRMSG file. The parameters are variables or literals to be used within the error message format. The general form of the STOP and ABORT statements is:

```
STOP {errnum{,param, param, ... }}

ABORT {errnum{,param, param, ... }}
```

Examples of the use of STOP, ABORT and END:

```
    *
    *
    *
    A=500
    B=750
    C=235
    D=1300
    REM COMPUTE PROFIT:
    REVENUE=A+B
    COST=C+D
    PROFIT=REVENUE-COST
    REM PRINT RESULTS
    IF PROFIT > 1 THEN GOTO 10
    PRINT "ZERO PROFIT OR LOSS"
    STOP <----------------------------------
10  PRINT "POSITIVE PROFIT"
    END <-----------------------------------
```

Explanation

If this path is taken, program will terminate.

Physical end of program.

```
    PRINT 'PLEASE ENTER FILE NAME':
    INPUT FN
    OPEN '', FN TO FFN ELSE ABORT 201, FN
    .
    .
    .
```

Explanation

This program requests a file name from the user and attempts to open the file. If an incorrect file name is entered, the standard system error message 201 "xxx IS NOT A FILE" will be printed and the program terminated.

4.7 PROGRAM SECURITY

4.7.1 BREAK AND ECHO COMMANDS

BREAK OFF may be used in a BASIC program to disallow the use of the BREAK key, which will prohibit users from entering the debugger. This may be cancelled by using a BREAK ON command within the program, or it will be automatically cancelled upon program termination.

These commands are cummulative. If two BREAK OFFs are executed, two BREAK ONs must be executed to restore a breakable status. The general form of the command is:

```
BREAK ON
BREAK OFF
```

The echoing of input on the terminal may be suppressed in a BASIC program by issuing the ECHO ON command. It may be turned back on again with the ECHO OFF. The general form of these commands are:

```
ECHO ON
ECHO OFF
```

Suppression of input to the terminal may be necessary when typing in sensitive material, such as passwords or personnel records.

subroutines and interprogram communication 5

5.1 INTERNAL SUBROUTINES

The GOSUB, COMPUTED GOSUB, RETURN, and RETURN TO statements provide internal subroutine capabilities for the BASIC program. A subroutine is an integral group of statements which handle a unique function or task. An internal subroutine is a subroutine that is contained within the program that calls it (i.e., it occurs before the END statement). The GOSUB statement transfers control to the subroutine. RETURN or RETURN TO statements return control to the main program.

5.1.1 GOSUB STATEMENT

The general form of the GOSUB statement:

```
GOSUB statement-label
```

Upon execution of a GOSUB statement, program control is transferred to the statement which begins with the specified numeric statement-label. Execution proceeds sequentially from that statement until a RETURN or RETURN TO statement is encountered. Either of these statements transfers control back to the main program.

5.1.2 COMPUTED GOSUB STATEMENT

The Computed GOSUB statement is a combination of the Computed GOTO statement and the GOSUB statement. Control is transferred to one of several statement labels selected by the current value of an index expression. Control returns to the statement following the computed GOSUB when a RETURN statement is executed. The general form of the Computed GOSUB statement:

```
ON expression GOSUB statement-label{,statement-label,...}
```

The expression is evaluated and truncated to an integer value. The result is used as an index into the list of statement-labels. A subroutine branch is executed to the statement-label selected.

If the expression evaluates to less than 1 or to a value greater than the number of statement-labels, no action is taken, that is, the statement immediately following the ON GOSUB will be executed next.

Examples of the use of GOSUB:

Correct Use		Explanation
	ON I GOSUB 100,150,200	
	. <----------	Control transfers here after return from subroutine (directly if I<1 OR I>3).
	.	
	.	
	.	
100	. <----------	Control transfers here if I=1.
	.	
	.	
	.	
	RETURN	
150	. <----------	Control transfers here if I=2.
	.	
	.	
	.	
	RETURN	
200	. <----------	Control transfers here if I=3.
	.	
	.	
	.	
	RETURN	

Incorrect Use	Explanation
ON GOSUB 100,200	Expression following the "ON" is missing.
ON A 100,200	"GOSUB" missing.

5.1.3 RETURN AND RETURN TO STATEMENTS

The general forms of the RETURN statement:

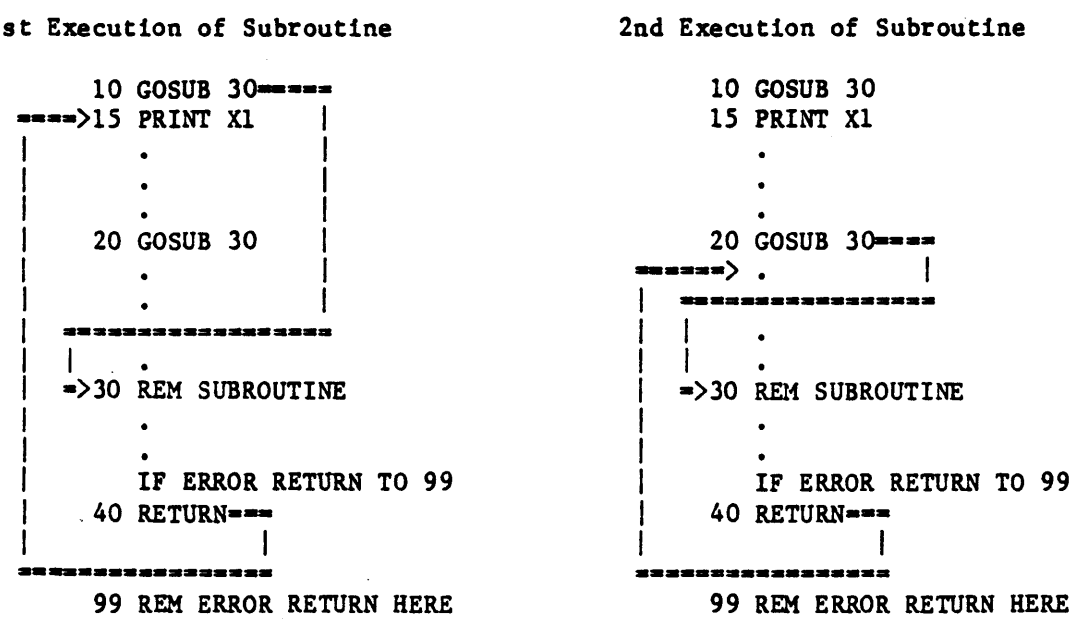
```
RETURN

RETURN TO statement-label
```

The RETURN statement will transfer control from the subroutine back to the statement immediately following the GOSUB statement. The RETURN TO statement returns control from the subroutine to the statement within the BASIC main program that has the specified statement-label.

The statements in a subroutine may be any BASIC statements, including another GOSUB statement. To ensure proper flow of control, each subroutine must return to the calling program by using a RETURN (or RETURN TO) statement, not a GOTO statement. Also, a subroutine should not be executed by any flow of control other than a GOSUB or ON GOSUB statement.

If the RETURN TO statement refers to a statement-label which is not present in the program, an error message will be printed at compile time (refer to Appendix B, COMPILER ERROR MESSAGES).

An example of correct use of subroutine statements:



In the example, when statement 10 is executed, control will transfer to statement 30 as illustrated in the left figure. The statements within the subroutine will be executed and statement 40 will then return control to statement 15. Execution will then proceed sequentially to statement 20, whereby control will again be transferred to the subroutine as shown in the right figure. The conditional RETURN TO path is taken instead of the normal RETURN if the logical variable ERROR is TRUE (=1).

The following example illustrates incorrect use of the BASIC subroutine capability.

Incorrect Use

```
A=1
GOSUB 100
A=A+1
GOTO 110     <-------   The GOTO statement should not be used to transfer
100 PRINT A             control to a subroutine.  This statement will
110 B=A*D               transfer control into the body of the subroutine,
PRINT B                 causing the "RETURN" to produce an error message:
RETURN                  "RETURN EXECUTED WITH NO GOSUB".
```

5.2 EXTERNAL SUBROUTINES

The CALL and SUBROUTINE statements provide external subroutine capbilities for the BASIC program. An external subroutine is a subroutine that is compiled and cataloged separately from the program or programs that call it.

5.2.1 CALL STATEMENT

The CALL statement has the following general form:

```
CALL name {(argument-list)}
```

The CALL statement transfers control to the cataloged subroutine 'name'. The CALL 'argument-list' consists of zero or more expressions, separated by commas, that represent actual values passed to the subroutine. The SUBROUTINE 'argument-list' consists of the same number of expressions, by which the subroutine references the values being passed to it.

5.2.2 SUBROUTINE STATEMENT

The SUBROUTINE statement has the following general form:

```
SUBROUTINE name {(argument-list)}
```

The SUBROUTINE statement is used to identify the program as a subroutine and must be the first statement in the program.

There is no correspondence between variable names or labels in the calling program and the subroutine. The only information passed between the calling program and subroutine are the arguments. A sample external subroutine that involves two arguments together with correctly formed CALL statements is:

CALL Statements	Subroutine ADD
CALL ADD (A,B,C)	SUBROUTINE ADD (X,Y,Z)
CALL ADD (A+2,F,X)	Z=X+Y
CALL ADD (3,495,Z)	RETURN
	END

An external subroutine must contain a SUBROUTINE statement, a RETURN statement and an END statement. GOSUB and RETURN may be used in the subroutine. When a RETURN is executed with no corresponding GOSUB, control passes to the statement following the corresponding CALL statement. If the subroutine's END statement, or a STOP or CHAIN statement is executed, control never returns to the calling program. The CHAIN statement should not be used to chain from an external subroutine to another BASIC program.

Examples of CALL and SUBROUTINE statements:

Correct Use	Explanation
CALL REVERSE (A,B) SUBROUTINE REVERSE (I,X)	Subroutine REVERSE has two arguments.
CALL REPORT SUBROUTINE REPORT	Subroutine REPORT has no parameters.
CALL VENDOR (NAME, ADDRESS, NUMBER) SUBROUTINE VENDOR (NAME, ADDR,NUM)	Subroutine VENDOR returns three values.
CALL DISPLAY (A,B,C) SUBROUTINE DISPLAY (I,J,K)	Subroutine DISPLAY accepts (and returns) three argument values.

Incorrect Use	Explanation
CALL	Subroutine name is missing.
CALL SUP (A B,C)	Comma is missing in argument list.
CALL COMP (X, Y, Z, RES) SUBROUTINE COMP (A, B, RES)	Number of arguments do not match.

5.2.3 ARRAY PASSING AND INDIRECT CALLS

Arrays may be passed to external subroutines. The general form for specifying an array in an argument list of CALL and SUBROUTINE statements is:

```
MAT variable
```

The 'variable' is the name of the array given in the DIMension statement. The array must be dimensioned in both the calling program and the subroutine. Array dimensions may be different, as long as the total number of elements matches. Arrays are copied in row major order (the rows are filled first). Consider the example:

Calling Program	Subroutine
DIM X(10), Y(10) CALL COPY (MAT X, MAT Y) END	SUBROUTINE COPY (MAT A) DIM A(10,2) PRINT A(15) RETURN END

In this subroutine, the parameter passing facility is used to copy MAT X and MAT Y specified in the CALL statement of the calling program into MAT A of the subroutine. Printing A(15) in the subroutine is equivalent to printing Y(5) in the calling program. Additional examples of array passing:

Correct Use	Explanation
DIM A(4,10),B(10,5) CALL REV (MAT A, MAT B)	Subroutine REV accepts two input array variables, one of size 40 and one of size 50 elements.
SUBROUTINE REV (MAT C, MAT B) DIM C(4,10), B(50)	

Incorrect Use	Explanation
DIM TAB (100) CALL SHORT (TAB)	The word 'MAT' must precede array TAB in the parameter list.
DIM FOUR (2,2) CALL GOF (MAT FOUR)	Corresponding arrays must have the same number of elements in the calling program and in the subroutine.
SUBROUTINE CAL(MAT NIX) DIM NIX(5)	

External subroutines may be called indirectly by using the indirect form of the CALL statement shown below:

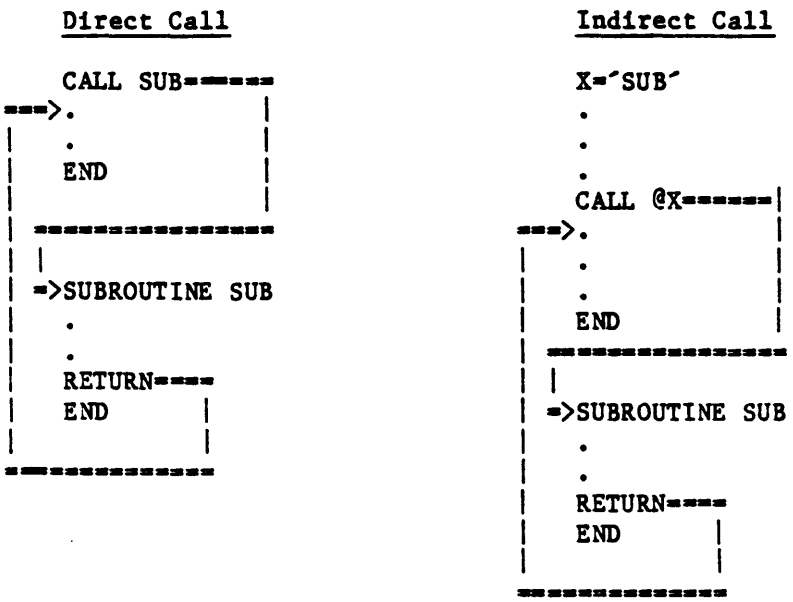
```
CALL @name {(argument list)}
```

The 'name' is a variable assigned to the cataloged subroutine to be called. The argument list performs the same function as in a direct call.

```
NAME = 'XSUB1'
CALL @NAME
NAME = 'XSUB2'
CALL @NAME
```

The first call invokes subroutine XSUB1. The second call invokes subroutine XSUB2.

The following example illustrates the difference between a direct call and an indirect call:



5.2.4 EXECUTE STATEMENT

The EXECUTE statement allows a BASIC program to temporarily suspend current operation in order to execute any command that may be entered at TCL and to use the results of that command later on in the BASIC program. Any TCL verbs or PICK statements, such as ACCESS input statements, SPOOLER printer control statements, utility verbs, as well as PROCs and cataloged BASIC programs may be executed. After execution of the command, the BASIC program will continue with the next statement following the EXECUTE statement.

EXECUTE statements may be serial or nested. There is no limit to the number of serial EXECUTEs within a BASIC program, but only up to 15 nested levels of EXECUTEs may be employed by a single user at one time. Also, the number of nested EXECUTEs must be within the range that has been preset by SYSPROG using the :TASKINIT verb (see Section 5.2.4.2 for details).

The EXECUTE statement may be used in two general forms:

```
EXECUTE expression {RETURNING variable-1} {CAPTURING variable-2}

EXECUTE expression {CAPTURING variable-2} {RETURNING variable-1}
```

where:

expression	is a complete PICK TCL statement, PROC, or cataloged BASIC program
variable-1	is a variable that will contain error message numbers after program execution (see Section 5.2.4.1.2).
variable-2	is a variable that will be used to capture the output from the executed command

RETURNING and CAPTURING phrases are optional. Both, either, or neither may be used. Note that TCL P verb has no effect when used with the CAPTURING clause.

5.2.4.1 EXECUTE Statement I/O

Input may be passed to the TCL command using the DATA statement. This is handled in the same manner as for the CHAIN statement. After the TCL command has been completed, the data stack will be reset.

Input may not be passed to or picked up from a PROC during an EXECUTE. (The EXECUTE turns off PROC stack.)

Output from the executed TCL command may be captured and placed in a variable in the calling BASIC program by using the CAPTURING variable-2 clause in the EXECUTE statement. If the command output is redirected back to the BASIC program, the output carriage-return/line-feed pairs will be converted to attribute marks and the clear-screen sequences to the terminal will be deleted.

5.2.4.1.1 Passing Select-Lists

If a select-list is generated, it may be passed back from the executed command to the BASIC program. The select-list may be assigned to the default SELECT statement select-variable or to a specified select-variable (by using the SELECT TO select-variable clause) for the next READNEXT statement.

If a select-list is active when the EXECUTE statement is executed, that list will be passed to the TCL command that is executed. You may choose to EXECUTE the SELECT verb, test for an active select-list, and then EXECUTE the SAVE-LIST verb. Or, you may issue a SELECT verb from TCL, run a BASIC program that EXECUTEs a LIST verb, and have the select-list that was generated at TCL passed to the LIST verb.

Note that once a select-list has been referenced by a BASIC SELECT or READNEXT statement, it may no longer be passed to another EXECUTEd TCL command.

5.2.4.1.2 Examining Error Message Numbers

You may examine error message numbers after the TCL command has been executed by including the RETURNING variable-1 clause in the EXECUTE statement. A variable will then be assigned to the returned error message numbers with each number separated by a blank.

You may also examine error message numbers by using the BASIC SYSTEM() function. The following statement will return the error message numbers for the executed TCL command with each message number separated by an attribute mark:

```
SYSTEM(17)
```

Note that the output format of the error message for a BASIC EXECUTE statement is the same as the error message format for a PROC. Therefore, when checking for EXECUTE error messages, there may be times when you need to use the FIELD statement in order to retrieve the desired error message. For example, after attaching a tape using T-ATT 4000, the error message buffer will contain both 90 and 4000. With a PROC you could use

```
IF E = 90 0 TAPE ATTACHED
```

to test the message. However, with the BASIC EXECUTE statement, you would first need to use the FIELD statement to extract the first field of the error message. For example:

```
EXECUTE 'T-ATT 4000' RETURNING ERRMSG
* ERRMSG WILL BE EQUAL TO 90 4000 IF TAPE IS ATTACHED
TATT.MSG = FIELD(ERRMSG,' ',1)
IF TATT.MSG = 90 THEN PRINT 'TAPE ATTACHED'
```

5.2.4.1.3 Determining Current Nested EXECUTE Level

If the executing BASIC program needs to determine the current level of nested EXECUTE statements (the statement that is now in process), the following function should be used: SYSTEM(16). This will return the current level number.

5.2.4.2 Allocating EXECUTE Workspaces and Nested EXECUTE Levels

The EXECUTE process requires separate workspaces in order to function. These workspaces are taken from the system Available Space (overflow) as needed and maintained in a special Execute Workspace Table for use by this process. Each workspace consists of 413 frames. When an EXECUTE statement is completed, whatever workspaces it used are returned to the Execute Workspace Table, not to system Available Space. These workspaces are then available for the next EXECUTE statement.

If a workspace can be taken from the Execute Workspace Table instead of the system Available Space, the EXECUTE process is not delayed. If workspaces need to be taken from the system Available Space, there may be a delay of up to 30 seconds.

In order to ensure maximum efficiency of the EXECUTE process, its workspaces may be preallocated. The :TASKINIT verb should be used to do this from the SYSPROG account. The format for this verb is shown below.

```
:TASKINIT  {workspaces}{,levels} {(U)}
```

where:

workspaces	is the number of EXECUTE workspaces to preallocate. If the workspace number is not given, no workspaces will be added to or deleted from the Execute Workspace Table. If the workspace number given is less than the currently allocated workspace number, then the extra workspaces will be returned to the system Available Space when the :TASKINIT verb is executed.
levels	is the maximum number of nested EXECUTEs in a process for a single user at one particular time. If this number is given, it must be in the range 0 to 15. A level of 0 will disallow use of the EXECUTE statement. If this number is omitted, the previous :TASKINIT level number is used, or if no previous :TASKINIT, the system default of 5 is used.
(U)	specifies that there should be unconditional reinitialization of the EXECUTE process. Note that this should only be used if there are severe problems in the EXECUTE environment that cannot be corrected without reinitialization. If the (U) option is used, only the port running the :TASKINIT verb should be logged on when the verb is executed. (It is imperative that no EXECUTE process be running when this option is used.)

5.2.4.3 Environment Changes After Using the EXECUTE Statement

The EXECUTE statement preserves the current working environment before executing a TCL statement, PROC or cataloged BASIC program, and restores it after the execution is completed. However, there are certain parameters that will not be restored if they have been altered by the EXECUTE statement when the BASIC program requesting the EXECUTE is resumed. These are listed below.

1. Terminal characteristics will not be restored if they have been changed by a TERM command.

2. A spooler assignment will not be restored if it has been altered by a spooler statement, such as SP-ASSIGN, SP-OPEN, SP-CLOSE, etc.

3. Current CPU usage charges will not be restored.

4. Tape attachment and tape record size will not be restored if they have been altered by use of any of the tape verbs.

5. If the OFF or LOGTO verb is used in an EXECUTE statement, the BASIC program that issued the EXECUTE statement will not be resumed. (When these verbs are used, the EXECUTE statement acts exactly like the CHAIN statement.)

6. If the Debugger was entered during the EXECUTE and an END or OFF command was used, the BASIC program that issued the EXECUTE system will not be resumed.

The following example prints item-ids of all PROCs in the Master Dictionary.

```
001 *BASIC
002 *EXECUTE STATEMENT, EXAMPLE 1
003 *
004 EQU TRUE TO 1, FALSE TO 0
005 OPEN 'MD' ELSE STOP 201,'MD'
006 EXECUTE 'SELECT MD IF 1 "PQ"' RETURNING ERRMSG
007 IF ERRMSG=401 THEN STOP; * [401] NO ITEMS PRESENT
008 EXECUTE 'SAVE-LIST PROC.LIST' CAPTURING OUTPUT RETURNING ERRMSG
009 IF ERRMSG=243 THEN;* [243] LIST SAVED
010   EXECUTE 'GET-LIST PROC.LIST' CAPTURING OUTPUT
011   IF SYSTEM(17)=202 THEN STOP; * [202] NOT ON FILE
012   EDI=FALSE
013   SELECT
014   LOOP
015     READNEXT ID ELSE EOI=TRUE
016   UNTIL EOI DO PRINT ID REPEAT
017 END
```

The example shown below prints the output of the WHAT verb using the CAPTURING clause.

```
001 *BASIC
002 *EXECUTE STATEMENT, EXAMPLE 2
003 *
004 EQU AM TO CHAR(254)
005 EXECUTE 'WHAT' CAPTURING OUTPUT RETURNING ERRMSG
006 NO.LINES=DCOUNT(OUTPUT,AM)
007 FOR LINE=1 TO NO.LINES
008   PRINT OUTPUT<LINE>
009 NEXT LINE
010 PRINT
011 PRINT 'ERRMSG = ':SYSTEM(17); * SEPARATED BY ATTRIBUTE MARKS
012 PRINT 'ERRMSG = ':ERRMSG; * SEPARATED BY SPACES
013 PRINT 'EXECUTE LEVEL = ':SYSTEM(16)
```

This example shows the use of the DATA statement to pass input to the TCL command. The DATA statement selects the Master Dictionary and lists the first five attributes of each item. The first five verbs it finds are then selected by the EXECUTE statement using a TCL SELECT command with the (n) option.

```
001 X='SELECT MD'
002 Y='LIST ONLY MD *A1 *A2 *A3 *A4 *A5'
003 DATA X,Y
004 EXECUTE 'SELECT MD IF 1 "P" (5)'
```

5.3 INTERPROGRAM COMMUNICATION

5.3.1 CHAIN STATEMENT

The CHAIN statement allows a BASIC program to execute any valid TCL command and gives it the ability to pass values to a separately compiled BASIC program which is executed during the same terminal session. The general form of the CHAIN statement is:

```
CHAIN "any TCL command"
```

The CHAIN statement causes the specified TCL command to be executed. The CHAIN statement may contain any valid Verb or PROC name in the user's Master Dictionary. Consider the following example:

```
CHAIN "RUN FILE1 PROGRAM1 (I)"
```

This statement causes the previously compiled program named PROGRAM1 in the file named FILE1 to be executed. The I option specifies that the data area is not to be reinitialized. This option must be used whenever values are to be passed from one program to another.

The CHAIN statement can allow values to be passed to the specified program since all BASIC programs which are executed during a single terminal session use the same data area. However, the variables in one program that are to be passed to another program must be in the same location. This is accomplished via use of the DIM statement. Consider the following two BASIC programs:

Program ABC in File BP

```
DIM A(1,1), B(2)
A(1,1)=500
B(1)=1 B(2)=2
CHAIN "RUN BP XYZ (I)"
END
```

Program XYZ in File BP

```
DIM I(2), J(1,1)
PRINT I(1),I(2),J(1,1)
END
```

Program ABC causes program XYZ to be executed. The I option used in the CHAIN statement specifies that the data area is not to be reinitialized, thus allowing program ABC to pass the values "500", "1", and "2" to program XYZ. Program XYZ, in turn, prints the values "500", "1", and "2". All dimensioned variables form a long vector in row major order, and on the chain are assigned left to right to the chained program's dimensioned variables.

Note that control is never returned to the BASIC program that originally executed the CHAIN statement.

Examples of the use of CHAIN:

Correct Use	Explanation
CHAIN "RUN FN1 LAX (I)"	Causes the execution of program LAX in file FN1. I option specifies that data area is not to be reinitialized (i.e., the program executing the CHAIN statement will pass values to program LAX).
CHAIN "LISTU"	Causes the execution of the LISTU SYSPROG PROC.
CHAIN "LIST MYFILE"	Causes the execution of the LIST ACCESS verb.
CHAIN "RUN PROGRAMS ABC"	Causes the execution of program ABC in file PROGRAMS. Since I option is not used, values will not be passed to program ABC.
A=LISTVERBS CHAIN A	Indirect form of CHAIN. Lists all verbs in user's dictionary on terminal.

Incorrect Use	Explanation
CHAIN	Parameter is missing.
CHAIN RUN BP ABC (I)	Quotes are missing around "RUN BP ABC (I)".
CHAIN=0	CHAIN cannot be used as a variable name.

5.3.2 DATA STATEMENT

The DATA statement is used to store data for queued input when using the CHAIN statement. The general form of the DATA statement is:

```
DATA expression
```

where 'expression' may be any valid combination of variables, literals, functions, etc.

Each DATA statement will generate one line of queued input. These input lines are then used in response to input requests from other processes. The DATA statement may be used to store queued input for ACCESS, TCL, PROCs, or other BASIC programs.

The following example illustrates the procedure to exit a BASIC program, sort select a file and begin execution of a second BASIC program. The variable REF-DATE is passed to the second BASIC program. Assuming that no queued input is currently present:

```
DATA 'RUN BP PROG'; DATA 'REF-DATE'

CHAIN 'SSELECT FILE WITH DATE  "':REF.DATE:'" BY DATE'
```

The first statement queues two values (e.g., 'RUN BP PROG' and 'REF-DATE'). The second statement causes an ACCESS statement to be executed. This is followed by input of the first value in the queue to the TCL prompt, beginning execution of BP PROG. Note that the queue is a First In First Out (FIFO) type, and therefore, the DATA statement must be processed before the CHAIN statement.

The second BASIC program (BP PROG) then performs the following:

```
INPUT REF-DATE
```

This instruction gets its input from the second value in the queue (i.e., the value of REF-DATE from the first BASIC program).

Multiple expressions are allowed on the DATA statement. Each expression becomes the response to one input request from the CHAINed process. Multiple DATA statements take the form:

```
DATA x,x,x, ...
```

Examples of the use of DATA:

Correct Use	Explanation
DATA A DATA B DATA C CHAIN 'RUN BP TEST'	Queues the values of A, B and C for subsequent input requests. Program 'TEST' may have three input requests which will be satisfied by the queued input.
DATA 'RUN BP CHARGE-ACC' DATA DATE CHAIN 'SELECT ACC WITH AMT > 100'	This causes the TCL command 'RUN BP CHARGE-ACC' to be stored in the queue. Control first exits to the ACCESS processor to perform the SELECT, after which the BASIC program is run with DATE as queued input.
DATA A,B,C	Multiple expressions may be queued by a single DATA statement.

Incorrect Use	Explanation
CHAIN 'RUN BP PROG' DATA X	The DATA statement must be processed before the CHAIN statement.

5.3.3 COMMON STATEMENT

The COMMON statement may be used to control the order in which space is allocated for the storage of variables and to pass values between programs. The general form of the COMMON statement is:

```
COM{MON} variable {,variable}...
```

The purpose of the COMMON statement is to change the automatic allocation sequence that the compiler follows, so that more than one program may have specified variables in a predetermined sequence.

In the absence of a COMMON statement, variables are allocated space in the order in which they appear in the program, with the additional restriction that arrays are allocated space after all simple variables. COMMON variables (including COMMON arrays) are allocated space before any other variables in the program. The COMMON statement must appear before any of the variables in the program are used.

The COMMON variable list may include simple variables, file variables and arrays. Arrays may be declared in a COMMON statement by specifying the dimensions enclosed in parentheses. For example, COMMON A(10) declares an array "A" with 10 elements. Arrays that are declared in a COMMON statement should not be declared again by a DIMENSION statement. All variables in the program which do not appear in a COMMON statement are allocated space in the normal manner.

The COMMON statement may be used to share variables among CHAINed programs, or among main-line programs and subroutines. This ensures that all 'COMMON' variables refer to the same stored values in different programs. Note that the 'I' option must be used with the RUN verb in chained programs to inhibit reinitialization. For example:

```
COMMON X,Y,Z(5)
COMMON Q,R,S(5)
```

If the first statement is found in a main-line program and the second in a subroutine call, the variables X and Q, Y and R, and the arrays Z and S will share the same locations. Note the second COMMON statement variables may be regarded as a mask over the first. What associates Q to X, R to Y, and S to Z is a matter of alignment. Thus, if the second statement had been "COMMON Q(2),R(5)" then Q(1) would refer to the location where the value of X is stored and Q(2) to the location where the value of Y is stored.

The COMMON statement differs from the argument list in a Subroutine Call in that the actual storage locations of COMMON variables are shared by the main-line program and its external subroutines; whereas the argument list in a Subroutine Call causes the values to be pushed onto the stack. The COMMON statement therefore provides a more efficient method of passing values.

Examples of the use of COMMON:

Correct Use	Explanation
Item "MAINPROG"	

`COMMON A,B,C(10)`
`A = "NUMBER"`
`B = "SQUARE ROOT"`
`FOR 1 = 1 TO 10`
`  C(I) = SQRT(I)`
`NEXT I`
	Variables A, B and array C are allocated space before any other variables.

`CALL SUB`
`PRINT "DONE"`
`END`
	Subroutine call to program SUBPROG.

Item SUBPROG	

`COMMON X(2),Y(10)`
`PRINT X(1), X(2)`
`FOR J = 1 TO 10`
`  PRINT J, Y(J)`
`NEXT J`
	The 2 elements of array X contain respectively, the values of A and B from the main-line program. The array Y contains the values of C from the main-line program.

`RETURN` `END`	Returns to main-line program.
Incorrect Use	**Explanation**
`COMMON A,B,C(10)` `DIM C(10)`	A DIMENSION statement should not be used for an array which has already been declared in a COMMON statement.

5.3.4 ENTER STATEMENT

The ENTER statement permits transfer of control from one cataloged program to another cataloged program. The program that executes the ENTER statement must be executed via the cataloged verb in the user's MD. The two forms of the ENTER statement are:

```
ENTER program-name
```

where program-name is the item-id of the program to be ENTERed and,

```
ENTER @variable
```

where variable has been assigned the program name to be ENTERed.

All variables which are to be passed between programs must be declared in a COMMON declaration in all program segments that are to be ENTERed. All other variables will be initialized upon ENTERing the program. It is permissible to ENTER a program that calls a subroutine, but it is illegal to ENTER a program from a subroutine.

Examples of the use of ENTER:

Correct Use	Explanation
`ENTER PROGRAM.1`	Causes execution of the cataloged program "PROGRAM.1". Any COMMON variables will be passed to "PROGRAM.1".

`N=2`
`PROG = "PROGRAM." : N`
`ENTER @PROG`
	Causes execution of the cataloged program "PROGRAM.2". Any COMMON variables will be passed to "PROGRAM.2".

Incorrect Use	**Explanation**
`ITEM="ABC"` `ENTER ITEM`	Would cause execution of the cataloged program "ITEM", not "ABC". Must be the indirect form (with "@") if program name is stored in a variable.
`ENTER=150`	ENTER cannot be used as a variable name.

intrinsic functions 6

6.1 NUMERIC FUNCTIONS

6.1.1 ABS

The ABS function generates the absolute numeric value of the expression. For example:

```
A=100
B=25
C=ABS(B-A)
```

These statements assign the value 75 to variable C.

6.1.2 INT

The INT function returns the integer portion of the specified expression. (The fractional portion of the expression is truncated after any operations have been performed.) For example:

```
PRINT INT(5.37)
```

This statement causes a value of 5 to be printed.

6.1.3 REM AND MOD

The REM and MOD functions are the same. They return the remainder of the value of the first expression divided by the value of the second expression. For example:

```
Q=REM(11,3)
```

This statement assigns the value 2 to variable Q.

6.1.4 SQRT

The general form of the SQRT function is:

```
SQRT(expression)
```

The SQRT or Square Root function returns the positive square root of any positive number (expression) that is greater than or equal to 0 and less than or equal to 41,073,748,835.5237. For example:

```
Y = SQRT(X)    0 <= X <= M    Assigns to Y the positive square root of the
                              positive number X.  Returns 0 if X < 0.
```

6.1.5 RND

The general form of the RND function is:

```
RND(expression)
```

The RND function generates a numeric value for a random number between zero and the number specified by the expression less one (inclusive), which must be positive. For example:

```
NUMBER = RND(201)
```

This statement generates a random number between 0 and 200 inclusive, and assigns its numeric value to the variable NUMBER.

Examples of the use of these functions:

Correct Use	Explanation
`Y = RND(X)`	Assigns to Y a random number between 0 and X.
`A = ABS(Q)`	Assigns the absolute value of variable Q to variable A.
`A = 600` `B = ABS(A-1000)`	Assigns the value 400 to variable B.

`A = 3.55`
`B = 3.6`
`C = INT(A+B)`
	Assigns the value 7 to variable C.

`J = INT(5/3)`	Assigns the value 1 to variable J.
`Z = RND(11)`	Assigns a random number between 0 and 10 (inclusive) to the variable Z.

`R = 100`
`Q = 50`
`B = RND(R+Q+1)`
	Assigns a random number between 0 and 150 (inclusive) to the variable B.

`Y = RND(ABS(051))`	Assigns a random number between 0 and 50 (inclusive) to the variable Y.
`Y = SQRT(36)`	Assigns the value 6 to variable Y.

Incorrect Use	Explanation
`Y = "ABCD"` `Z = ABS(Y)`	Expression in ABS function must be numeric.
`X = RND (101)`	A space is not allowed between "RND" and "(".

6.2 TRIGONOMETRIC FUNCTIONS

Trigonometric functions included in BASIC are SINE, COSINE, TANGENT, NATURAL LOGARITHM, EXPONENTIAL, and POWER. The SINE, COSINE, and TANGENT functions return the function of an angle expressed in degrees. In this section, M is used to denote the integer 14,073,748,835.5327, which is the largest allowable number in BASIC.

6.2.1 COSINE

The general form of the COSINE function is:

```
COS(expression)
```

The COSINE function returns the cosine of an angle expressed in degrees. The given angle must be less than or equal to M and greater than or equal to -M.

6.2.2 SINE

The general form of the SINE function is:

```
SIN(expression)
```

To generate the sine of an angle expressed in degrees, the SINE function is used. The given angle must be less than or equal to M and greater than or equal to -M.

6.2.3 TANGENT

The general form of the TANGENT function is:

```
TAN(expression)
```

The TANGENT function will produce the tangent of an angle expressed in degrees. The angle must be less than or equal to M and greater than or equal to -M.

6.2.4 LOGARITHM

The general form of the NATURAL LOGARITHM function is:

```
LN(expression)
```

The LN function generates the natural (base e) logarithm of the expression. If the value of the expression is less than or equal to zero, the LN function returns a value of zero.

6.2.5 EXPONENTIAL

The general form of the EXPONENTIAL function is:

EXP(expression)

The EXPONENTIAL function raises the number 'e' (2.7183) to the value of the expression. The EXPONENTIAL function is the inverse of the NATURAL LOGARITHM (LN) function. If the value of the expression is such that 'e' to that power is greater than M, the function returns a value of zero.

6.2.6 POWER

The general form of the POWER function is:

PWR(expression,expression)

The POWER function raises the first expression to the power denoted by the second expression. If the second expression is zero, the function will return the value one. Like the EXP function, if the first expression raised to the power denoted by the second expression is greater than M, the function will return unpredictable numbers. If the first expression is zero and the second expression is any number other than zero, the function will return a value of zero. Another way to express the PWR function is X^Y where X is raised to the Y power.

A summary of trigonometric functions and the acceptable range of expressions is shown below. In this summary, M is used to denote the integer 14,073,748,835.5327, which is the largest allowable number in BASIC.

Function	Range of X	Description
COS(X)	-M <= X <= M	Returns the cosine of an angle of X degrees.
SIN(X)	-M <= X <= M	Returns the sine of an angle of X degrees.
TAN(X)	-M <= X <= M -M <= RESULT <= M	Returns the tangent of an angle of X degrees.
LN(X)	0 < X <= M -M <= RESULT <= M	Returns the natural (base e) logarithm of the expression X.
EXP(X)	-M <= RESULT <= M	Raises the number 'e' (2.7183) to the value of X.
PWR(X,Y)	-M <= RESULT <= M	Raises the first expression to the power denoted by the second expression.

6.3 LOGICAL FUNCTIONS

6.3.1 NOT

The general form of the NOT functions is:

```
NOT(expression)
```

The NOT function returns the logical inverse of the specified expression; it returns a value of TRUE (generates a value of 1) if the expression evaluates to 0, and returns a value of FALSE (generates a value of 0) if the expression evaluates to a non-zero quantity. The specified expression must evaluate to a numeric quantity or a numeric string. The following statement, for example, assigns the value of 1 to the variable X:

```
X = NOT(0)
```

As a further example, the following statements cause the value 0 to be printed:

```
A = 1
B = 5
PRINT NOT(A AND B)
```

6.3.2 NUM AND ALPHA

The general form of the NUM and ALPHA functions is:

```
NUM(expression)
ALPHA(expression)
```

The NUM function tests the given expression for a numeric value and the ALPHA tests for an alphabetic value. For example, if the expression evaluates to a number or numeric string, the NUM function will return a value of TRUE (i.e., generate a value of 1), while the ALPHA function will return a value of FALSE (generate a value of 0). Inversely, an expression evaluating to a letter or an alphabetic string will cause the NUM function to return a value of FALSE, while the ALPHA function will return a value of TRUE. Consider the following:

```
IF NUM(expression) THEN PRINT "NUMERIC DATA"
```

This statement will print the text "NUMERIC DATA" if the current value of variable "expression" is a number or a numeric string.

```
IF ALPHA(expression) THEN PRINT "ALPHABETIC DATA"
```

This statement will print the text "ALPHABETIC DATA" if the current value of variable "expression" is a letter or an alphabetic string. In the case of a non-numeric, non-alphabetic character or string (i.e., #, ?, etc.) a value of FALSE would be returned for both functions. The empty string ("") is considered to be a numeric string, but not an alpha string.

Examples of the use of logical functions:

Correct Use	Explanation
X=A AND NOT(B)	Assigns the value 1 to variable X if current value of variable A is 1 and current value of variable B is 0. Assigns a value of 0 to X otherwise.
IF NOT(X1)THEN STOP	Program terminates if current value of variable X1 is 0.
A1=NUM(123)	Assigns a value of 1 to variable A1.
A2=NUM("123")	Assigns a value of 1 to variable A2.
A3=NUM("12C")	Assigns a value of 0 to variable A3.
IF ALPHA(I CAT J) THEN GOTO 5	Transfers control to statement 5 if current value of both variables I and J are letters or alphabetic strings.
PRINT NOT(M) OR NOT(NUM(N))	Prints a value of 1 if current value of variable M is 0 or current value of variable N is a non-numeric string. Otherwise prints a zero.

Incorrect Use	Explanation
PRINT NOT A	Parentheses are missing around variable A.
NUM(X)=5	Intrinsic functions may not appear on the left side of the equality sign.
IF NUM( ) THEN STOP	Expression is missing.

file handling 7

7.1 FILE SELECTION FOR I/O

7.1.1 OPEN STATEMENT

The OPEN statement is used to select a PICK file for subsequent input, output or update.

Before a PICK file can be accessed by a READ, WRITE, DELETE, MATREAD, MATWRITE, READV, or WRITEV, etc., statement, it must be opened via an OPEN statement. The general form of the OPEN statement is:

```
OPEN {"DICT",} "file-name" {TO file-variable} THEN/ELSE statements
```

The second expression in the OPEN statement indicates the PICK file name. If the first expression is "DICT", then the dictionary section of the file is opened. (The word DICT must be explicitly supplied to open a dictionary level file.) Note that either single or double quotes may be used in the statement. Consider the following statements:

```
OPEN 'file-name' THEN/ELSE statements
OPEN '','file-name' THEN/ELSE statements
```

They are equivalent since the leading null expression is optional. In both cases, the data section is opened. If the file is a multiple data file (that is, multiple data files associated with a single dictionary), to open one of the data sections, the format used is:

```
"dict-name,data-name"        or        " ","dict-name,data-name"
```

If the "TO file-variable" option is used, then the dictionary or data section of the file will be assigned to the specified variable for subsequent reference. If the "TO file-variable" option is omitted, then an internal default file-variable is generated; subsequent I/O statements not specifying a file-variable will then automatically default to this file.

Depending on whether the PICK file indicated in the OPEN statement exists, the statement or sequence of statements following the THEN/ELSE will be executed. The statements in the THEN/ELSE clause may be placed on the same line separated by semicolons, or may be placed on multiple lines terminated by an END. The THEN/ELSE clause follows the same format as the THEN/ELSE clause in the IF statement.

There is no limit to the number of files that may be open at any given time. Consider the following example:

```
OPEN "DICT","QA4" TO F1 ELSE PRINT "NO FILE"; STOP
```

This statement will open the dictionary portion of the file named QA4 and will assign it to file-variable F1. If QA4 does not exist, the message "NO FILE" will be printed and the program will terminate. The data portion of a file named TEST is opened as illustrated below:

```
OPEN 'TEST' ELSE
    PRINT "TEST DOES NOT EXIST"
    GOTO 100
    END
```

In this example, the file is assigned to an internal default file-variable. The message "TEST DOES NOT EXIST" will be printed and control will pass to statement 100 if the file named TEST does not exist. Examples of OPEN:

Correct Use	Explanation

`A='DICT'`
`OPEN A, 'XYZ' TO B ELSE`
`  PRINT "NO XYZ"`
`  STOP`
`  END`
	Opens the dictionary portion of file XYZ and assigns it to file-variable B. If XYZ does not exist, the text "NO XYZ" is printed and the program terminates.

`OPEN 'ABC,X' TO D5 ELSE` `STOP`	Opens data section X of file ABC and assigns it to file-variable D5. If ABC,X does not exist, program terminates.

`X=''`
`Y='TEST1'`
`Z='MY FILE'`
`OPEN X, Y THEN PRINT Z`
`GOTO 5`
	Opens data section of file TEST1 and assigns it to internal default file-variable. If TEST1 exists, "MY FILE" is printed and control passes to statement 5.

Incorrect Use	Explanation
`OPEN 'DICT' TO Q ELSE STOP`	Second expression (file name) is missing.
`OPEN '','ABC'`	THEN/ELSE clause is missing.
`OPEN "F", "D#" ELSE STOP`	First expression must be 'DICT' or ''.

`OPEN 'INC' TO C1 ELSE`
`  PRINT 'NO FILE'`
`  X=Y+1`
`  GOTO 50`
	END is missing after the statements in the ELSE clause.

7.2 CLEARING A FILE

7.2.1 CLEARFILE STATEMENT

The CLEARFILE statement is used to clear out the data section of a specified file. The general form of the CLEARFILE statement is:

```
CLEARFILE {file-variable}
```

Upon execution of the CLEARFILE statement, the data section of the file which was previously assigned to the specified file variable via an OPEN statement will be emptied. The data in the file will be deleted, but the file itself will not be deleted. If the file variable is omitted from the CLEARFILE statement, then the internal default variable is used (thus specifying the file most recently opened without a file-variable).

Consider the following example:

```
OPEN 'AFILE' TO X ELSE PRINT "CANNOT OPEN"; STOP
CLEARFILE X
```

These statements cause the data section of the file named AFILE to be cleared.

The dictionary section of file cannot be cleared via a CLEARFILE statement.

Note that the BASIC program will abort with an appropriate error message if the specified file has not been opened prior to the execution of the CLEARFILE statement (refer to Appendix C, RUN-TIME ERROR MESSAGES).

Examples of the use of CLEARFILE:

Correct Use	Explanation
OPEN 'FN1' ELSE PRINT 'NO FN1';STOP READ I FROM 'I1'ELSE STOP CLEARFILE	Opens the data section of file FN1, reads item I1 and assigns value to variable I, and finally clears the data section of file FN1.
OPEN 'FILEA' TO A ELSE STOP OPEN 'FILEB' TO B ELSE STOP CLEARFILE A CLEARFILE B	Clears the data sections of files FILEA and FILEB.
OPEN 'ABC' ELSE PRINT 'NO FILE';STOP READV Q FROM 'IB3', 5 ELSE STOP IF Q='TEST' THEN CLEARFILE	Clears the data section of file ABC if the 5th attribute of the item with name IB3 has a string value of 'TEST'.

Incorrect Use	Explanation
CLEARFILE A+B	A+B is not a legal variable.
CLEARFILE A,B	Only one file can be cleared per CLEARFILE statement.
OPEN 'DICT','F5' TO C ELSE STOP CLEARFILE C	The dictionary section of a file cannot be cleared via a CLEARFILE statement.

7.3 ACCESSING FILE ITEMS

7.3.1 READ STATEMENT

The READ statement reads a file item and assigns its value to a variable. The READ statement has the following general form:

```
READ variable FROM {file-variable,} item-name THEN/ELSE statements
```

The READ statement reads the file item specified by item-name and assigns its string value to the first variable. The file-variable is optional; if used, the item will be read from the file previously assigned to file-variable via an OPEN statement. If file-variable is omitted, then the internal default variable is used (thus specifying the file most recently opened without a file- variable).

Depending on whether the item-name specifies the name of an item which exists, the statement or sequence of statements following the THEN/ELSE will be executed. The statements in the THEN/ELSE clause may appear on one line separated by semicolons, or on multiple lines terminated by an END. The THEN/ELSE clause takes on the same format as the THEN/ELSE clause in the IF statement). Consider the example:

```
READ X1 FROM W,"TEMP" ELSE PRINT "NON-EXISTENT"; STOP
```

This statement will read the item named TEMP from the file opened and assigned to file-variable W, and will assign its string value to variable X1; program control will then pass to the next sequential statement in the program. If the file item TEMP does not exist, the message "NON-EXISTENT" will be printed and the program will terminate.

Note that the item-name should be surrounded by quotes if it directly references an item, but that quotes are not needed if an indirect reference is used.

Note that the BASIC program will abort with an appropriate error message if the specified file has not been opened prior to the execution of the READ statement (refer to Appendix C, RUN-TIME ERROR MESSAGES).

Examples of the use of READ:

Correct Use	Explanation
READ A1 FROM X,"ABC" THEN PRINT "ABC" GOTO 70 END	Reads item ABC from the file opened and assigned to file variable X, and assigns it value to variable A1. If ABC exists, the text "ABC" is printed and control passes to statement 70.
A="TEST" B="1" READ X FROM C,(A CAT B) ELSE STOP	Reads item TEST1 from the file opened and assigned to file variable C, and assigns its value to variable X. Program terminates if TEST1 does not exist.
READ Z FROM "Q" ELSE PRINT X; STOP	Reads item Q from file opened without a file variable and assigns its value to variable Z. Prints value of X and terminates program if Q does not exist.

Incorrect Use	Explanation
READ X1 ELSE STOP	"FROM expression" is missing.
READ A+B FROM I, "CB" ELSE STOP	A+B is not a legal variable name.
READ VAR FROM X, "ABC"	THEN/ELSE clause is missing.

7.3.2 SELECT STATEMENT

The SELECT statement allows you to select a set of item-ids or attributes which, when used in conjunction with the READNEXT statement, may be used to access single or multiple file item-ids or attributes within a BASIC program. The general form of the SELECT statement is:

```
SELECT {file-variable}{TO select-variable}
```

The SELECT statement builds the same list of item-ids as an ACCESS SELECT statement executed at the terminal without any selection criteria. If the file-variable is used, a list of item-ids will be created for the file or item previously assigned to file-variable via an OPEN or READ statement. If the file-variable is omitted, then the internal default variable is used (thus specifying the file most recently opened without a file-variable). For example, the following BASIC program will print the item-ids in the file named BP. This is equivalent to the ACCESS command 'SELECT BP' executed on the terminal:

```
     OPEN 'BP' ELSE STOP
     SELECT
10   READNEXT ID ELSE STOP
     PRINT ID
     GOTO 10
```

There are six forms of the SELECT statement:

1. SELECT
   Creates a list of item-ids from the file most recently opened without a file-variable.

2. SELECT file-variable
   Creates a select list of item-ids from the file opened to 'file-variable'.

3. SELECT var
   Creates a select list from the attributes of the variable 'var'. The select list will only include the first value of a multi-valued attribute.

4. SELECT TO select-variable
   Creates a select list from the file most recently opened without a file variable and assigns the selected list to 'select-variable'.

5. SELECT file-variable TO select-variable
   Creates a select list from the file opened to 'file-variable' and assigns the selected list to 'select-variable'.

6. SELECT var TO select-variable
   Creates a select list from the attributes of the variable 'var' and assigns the selected list to 'select-variable'.

Examples of the use of SELECT:

Correct Use	Explanation
SELECT	Builds list of item-ids using the default variable of the last file opened without a file-variable.
SELECT BP TO BLIST	Builds a list of item-ids for the file opened and assigned to file-variable 'BP'. Assigns the list to select-variable 'BLIST'.
READ A FROM FILEX,'ALIST' ELSE STOP SELECT A	Creates a select list of the attributes in item ALIST.

Incorrect Use	Explanation
SELECT A+B	A variable name (not an expression) must be used in the SELECT statement.

7.3.3 READNEXT STATEMENT

The READNEXT statement reads the next item-id from a selected list. If multiple files have been selected, which list to read is specified by the select variable. The general form of the READNEXT statement is:

```
READNEXT variable {,vmc}{FROM select-variable} THEN/ELSE statements
```

The four possible forms of the READNEXT are:

1. READNEXT variable THEN/ELSE statements
   This will read the next Item-id of the last file selected without a select-variable.

2. READNEXT variable,vmc THEN/ELSE statements
   Here 'vmc' is used for the value mark count to be obtained from the Exploding Sort (External SSELECT).

3. READNEXT variable FROM select-variable THEN/ELSE statements
   This reads the next item-id of the file (or variable) selected and assigned to the select variable.

4. READNEXT variable,vmc FROM select-variable THEN/ELSE statements
   This is a combination of the previous two forms.

The READNEXT statement reads the next item-id and assigns its string value to the variable indicated. The item-id is read from the list created by the most recent program SELECT statement or SELECT, SSELECT, or QSELECT command issued before the BASIC program execution. Depending on whether or not the list of item-ids has been exhausted, or if selection has been performed, the statements following the THEN/ELSE will be executed. The statements in the THEN/ELSE clause may be placed on the same line separated by semicolons, or may be placed on multiple lines terminated by an END. The THEN/ELSE clause takes on the same format as the THEN/ELSE clause in the IF statement.

Consider the following example:

```
READNEXT VAR1 ELSE PRINT "CANNOT READ"; GOTO 10
```

This statement will read the next item-id and assign its string value to the variable VAR1. If the list of item-ids has been exhausted (or if a program SELECT statement, or a SELECT, SSELECT, or QSELECT command has not been issued before the BASIC program execution), then the message "CANNOT READ" will be printed and control will pass to statement 10.

Examples of the use of READNEXT:

Correct Use	Explanation
READNEXT A FROM X ELSE STOP	Specifies the list selected and assigned to the select-variable X. Assigns the value of that list's next item-id to variable A. If item-id list is exhausted (or if no SELECT, SSELECT or QSELECT executed), program will terminate.
READNEXT X2 ELSE PRINT "UNABLE" GOTO 50 END	Specifies the last list selected without a select-variable. Assigns the value of the next item-id to variable X2. If unable to read, "UNABLE" is printed and control transfers to statement 50.
FOR X=1 TO 10 READNEXT B(X) ELSE STOP NEXT X	Reads next ten item-ids and assigns values to matrix elements B(1) through B(10).

Incorrect Use	Explanation
READNEXT Q5	THEN/ELSE clause is missing.
READNEXT Z THEN	At least one statement must follow THEN.
READNEXT ELSE GOTO 500	Variable is missing.

7.4 MODIFYING AND DELETING FILE ITEMS

7.4.1 WRITE STATEMENT

The WRITE statement is used to update a file item. The general form of the WRITE statement:

```
WRITE expression ON {file-variable,} item-name
```

The WRITE statement replaces the content of the item specified by item-name with the string value of expression. The file-variable is optional; if used, the item will be replaced in the file previously assigned to file-variable via an OPEN statement. If the file-variable is omitted, the internal default variable is used (thus specifying the file most recently opened without a file variable). If item-name specifies an item which does not exist, then a new item will be created.

The following statements, for example, replace the current content of the item named XYZ in the file opened and assigned to file-variable F5 with the string value "THIS IS AN EXAMPLE":

```
VALUE = "THIS IS AN EXAMPLE"
WRITE VALUE ON F5,"XYZ"
```

Alternatively, this example may have been specified as follows:

```
WRITE "THIS IS AN EXAMPLE" ON F5,"XYZ"
```

7.4.2 DELETE STATEMENT

The DELETE statement is used to delete a file item. The general form of the DELETE statement is:

```
DELETE {file-variable,} item-name
```

The DELETE statement deletes the item which is specified by item-name and is located in the file previously assigned to the specified file-variable via an OPEN statement. If the file-variable is omitted, then the internal default variable is used (thus specifying the file most recently opened without a file-variable). For example:

```
DELETE  AB,"TESTITEM"
```

This statement will delete the item named TESTITEM in the file previously opened and assigned to file-variable AB.

No action is taken if a non-existent item is specified in the DELETE statement. Note that if the item is directly referenced, the item-name must be in quotes.

The BASIC program will abort with an appropriate error message if the specified file has not been opened prior to the execution of the WRITE or DELETE statement.

Examples of the use of WRITE and DELETE:

Correct Use	Explanation
WRITE "XXX" ON A, "ITEM5"	Replaces the current content of item ITEM5 (in the file opened and assigned to variable A) with string value "XXX".
A="123456789" B="X55" WRITE A ON FN1,B	Replaces the current content of item X55 (in the file opened and assigned to variable FN1) with string value "123456789".
WRITE 100*5 ON "EXP"	Replaces the current content of item EXP (in the file opened without a file variable) with string value "500".
DELETE X, "XYZ"	Deletes item XYZ in the file opened and assigned to variable X.
Q="JOB" DELETE Q	Deletes item JOB in the file opened without a file variable.

Incorrect Use	Explanation
WRITE "BBBB" ON F,	Second expression is missing.
WRITE ON B,"XYZ"	First expression is missing.
DELETE X, "5", ELSE STOP	ELSE clause not allowed.

7.5 ACCESSING AND UPDATING SINGLE ATTRIBUTES

7.5.1 READV STATEMENT

The READV statement is used to read a single attribute value from an item in a file. The general form of the READV statement is:

```
READV variable FROM {file-variable,} item-name, attribute-number
THEN/ELSE statements
```

The READV statement reads the attribute-number from the item-name specified and assigns its string value to the first variable.

The file-variable is optional; if it is used, the attribute will read from the file previously assigned to that variable via an OPEN statement. If the file-variable is omitted, then the internal default variable is used (thus specifying the file most recently opened without a file variable).

If a non-existent item is specified, the statement or sequence of statements following an ELSE will be executed; otherwise, the statement(s) following a THEN will be performed. The statements in the THEN/ELSE clause may be placed on the same line separated by semicolons, or may be placed on multiple lines terminated by END. The THEN/ELSE clause takes on the same format as the THEN/ELSE clause in the IF statement).

Consider the following example:

```
READV A FROM F,"XYZ", 3 ELSE STOP
```

This statement reads the third attribute of item XYZ (in the file opened and assigned to file-variable F) and assigns its value to variable A. If item XYZ does not exist, the program terminates.

The BASIC program will abort with an appropriate error message if the specified file has not been opened prior to the execution of the READV statement.

Examples of the use of READV:

Correct Use	Explanation
READV X FROM A, "TEST", 5 ELSE PRINT ERR GOTO 70 END	Reads 5th attribute of item TEST (in the file opened and assigned to variable A) and assigns value to variable X. If item TEST is non-existent, then value of ERR is printed and control passes to statement 70.

Incorrect Use	Explanation
READV X*Y FROM F,"B",2 ELSE STOP	A variable name (not an expression) must appear between READV and FROM.
READV A FROM B,"Z3" THEN STOP	Attribute number is missing.
READV B FROM "XYZ",A+Q	ELSE clause is missing.

7.5.2 WRITEV STATEMENT

The WRITEV statement writes a single attribute value to an item in a file. It is used to update attribute values. Its general form is:

```
WRITEV expression ON {file-variable,} item-name, attribute-number
```

Upon execution of the WRITEV statement, the value of the first expression becomes the attribute specified by attribute number in the item specified by item-name and in the file previously assigned to the specified file-variable via an OPEN statement.

If file-variable is omitted, then the internal default variable will be used (thus specifying the file most recently opened without a file-variable).

If a non-existent item name or attribute number is specified, then a new item or attribute will be created. Consider the example:

```
X1 = "XXX"
WRITEV X1 ON A2,"ABC",4
```

These statements replace the 4th attribute of item ABC (in the file opened and assigned to variable A2) with the string value "XXX".

The WRITEV statement will also allow the attribute mark count to have a value of either zero or minus one, thus inserting data before the first attribute or following the last attribute. For example:

```
WRITEV XX ON FILE, "ITEM", AMC
```

When AMC=0, the attribute XX is inserted at the beginning of the item ITEM. All attributes in the item are shifted by 1 attribute and the attribute XX becomes attribute 1.

When AMC=-1, the attribute XX is appended to the end of the item ITEM. The number of attributes in the item increase by 1 and all previously existing attributes are undisturbed.

The BASIC program will abort with an appropriate error message if the specified file has not been opened prior to the execution of the WRITEV statement.

Examples of the use of WRITEV:

Correct Use	Explanation
Y="THIS IS A TEST" WRITEV Y ON X,"PROG",0	The string value "THIS IS A TEST" is inserted before the first attribute of item PROG in the file opened and assigned to variable X.
WRITEV "XYZ" ON "A7",4	Attribute 4 of item A7 (in the file opened without a file variable) is replaced by string value "XYZ".

Incorrect Use	Explanation
WRITEV I ON "ABC" J	Comma is missing between "ABC" and J.

7.6 ACCESSING AND UPDATING MULTIPLE ATTRIBUTES

7.6.1 MATREAD STATEMENT

The MATREAD statement reads a file item and assigns the value of each attribute to consecutive vector elements. The MATREAD statement has the following form:

```
MATREAD array-variable FROM {file-variable,} item-name THEN/ELSE statements
```

The MATREAD statement reads the file item specified by item-name and assigns the string value of each attribute to consecutive elements of the vector specified by the array-variable. If the file-variable is used, the item will read from the file previously assigned to file-variable via an OPEN statement. If file-variable is omitted, then the internal default variable is used (thus specifying the file most recently opened without a file variable).

If a non-existent item is specified, then the statements following the ELSE will be executed; otherwise, statements following the THEN will be performed. The statements in the THEN/ELSE clause may appear on one line separated by semicolons, or on multiple lines terminated by an END. The THEN/ELSE clause takes on the same format as the THEN/ELSE clause in the IF statement. If the item does not exist, the contents of the vector remain unchanged. Consider:

```
MATREAD IN FROM 'ITEM' ELSE STOP
```

This statement will read into array "IN" the item named "ITEM" from the file most recently opened without a file variable. If ITEM does not exist, the program stops. If the number of attributes in the item is less than the DIMensioned size of the vector, the trailing vector elements are assigned a null string. If the number of attributes in the item exceeds the DIMensioned size of the vector, the remaining attributes will be assigned to the last element of the array.

7.6.2 MATWRITE STATEMENT

The MATWRITE statement writes a file item with the contents of a vector. The MATWRITE statement has the following general form:

```
MATWRITE array-variable ON {file-variable,} item-name
```

The MATWRITE statement replaces the attributes of the item specified by item-name with the string value of the consecutive elements of the vector named by the array-variable. If the file-variable is used, the item will be written in the file previously assigned to file-variable via an OPEN statement. If the file-variable is omitted, then the internal default variable is used. If the item-name specifies an item which does not exist, then a new item will be created. The number of attributes in the item is determined by the DIMensioned size of the vector.

For example:

```
MATWRITE IN ON 'ITEM'
```

This statement will write the contents of array "IN" to the default file as item "ITEM".

Examples of the use of MATREAD and MATWRITE:

Correct Use	Explanation
DIM ITEM (20) OPEN '', 'LOG' TO F1 ELSE STOP MATREAD ITEM FROM F1, 'TEST' ELSE STOP	Reads the item named TEST from the data file named LOG and assigns the string value of each attribute to consecutive elements of vector ITEM, starting with the first element.
DIM ITEM (10) OPEN '', 'TEST' ELSE STOP FOR I=1 TO 10 ITEM(I)=I NEXT I MATWRITE ITEM ON "JUNK"	Writes an item named JUNK in the file named TEST. The item written will contain 10 attributes whose string values are 1 through 10.

Incorrect Use	Explanation
MATREAD	Parameters missing.
MATREAD FROM A THEN GOTO 10	First variable is missing.
MATREAD X ON F1 ELSE STOP	Word should be FROM, not ON.
MATWRITE M ON FL+1, 'ITEM'	File name following ON must be a variable, not an expression.

7.7 MULTIUSER FILE AND EXECUTION LOCKS

7.7.1 BASIC LOCKS

The LOCK and UNLOCK statements provide a file and execution lock capability for BASIC programs. The LOCK statement sets execution locks while the UNLOCK statement releases them.

7.7.1.1 LOCK STATEMENT

The LOCK statement sets an execution lock so that when any other BASIC program attempts to set the same lock, then that program will either execute an alternate set of statements or will pause until the lock is released (via an UNLOCK statement) by the program which originally locked it. Execution locks may be used as file locks to prevent multiple BASIC programs from updating the same files simultaneously. There are 48 execution locks numbered from 0 through 47. The LOCK statement has the following general form:

```
LOCK lock-number {ELSE statements}
```

The value of the lock-number specifies which execution lock is to be set. If the specified execution lock has already been set by another concurrently running program (and the ELSE clause is not used), then program execution will temporarily halt until the lock is released by the other program. If the ELSE clause is used, then the statement(s) following the ELSE will be executed if the specified lock has already been set by another program. The statements in the ELSE clause may be placed on the same line separated by semicolons, or may be placed on multiple lines terminated by an END. The ELSE clause takes on the same format as the ELSE clause in the IF statement.

7.7.1.2 UNLOCK STATEMENT

The UNLOCK statement has the following general form:

```
UNLOCK {lock-number}
```

The value of the lock-number specifies which execution lock is to be released (cleared). If the number is omitted, then all execution locks which were previously set by the program will be released.

All execution locks set by a program will automatically be released upon termination of the program.

As an overall example of the execution lock capability, consider the following situation. Process A sets execution lock 42 before executing a section of code that should be non-reentrant (that is, code which should not be executed by more than one process simultaneously). Process B executing the same program reaches the "LOCK 42" instruction, but cannot lock that section of code until Process A has unlocked 42. This has made the code non-reentrant.

Examples of the use of LOCK and UNLOCK:

Correct Use	Explanation
LOCK 15 ELSE STOP	Sets execution lock 15 (if lock 15 is already set, program will terminate.
LOCK 2	Sets execution lock 2.
LOCK 10 ELSE PRINT X; GOTO 5	Sets execution lock 10 (if lock 10 is already set, the value of X will be printed and the program will branch to statment 5).
UNLOCK 63	Resets execution lock 63.
UNLOCK	Resets all execution locks previously set by the program.
UNLOCK (5+A)*(8-2)	The current value of the expression (5+A)*(B-2) specifies which execution lock is released.

Incorrect Use	Explanation
LOCK	Expression is missing.
LOCK 3,21	Only one lock may be set per LOCK. statement.
UNLOCK (5+A)(B-2)	Expression is illegal.

7.7.2 READ WITH LOCK FOR UPDATING: READU, READVU, AND MATREADU STATEMENTS

The READU, READVU, and MATREADU statements allow you to lock a group of items in a file prior to updating an item in the group. Using a group lock prevents updating of an item by two or more programs simultaneously while still allowing multiple program access to the file. These statements have the following general form:

READU variable FROM {file-variable,} item-name THEN/ELSE statements

READVU variable FROM {file-variable,} item-name, attribute-number THEN/ELSE statements

MATREADU array-variable FROM {file-variable,} item-name THEN/ELSE statements

These statements function in the same way as the READ, READV, and MATREAD statements, but in addition lock the group of the file in which the item to be accessed falls. A group lock will prevent:

1. Access of items in the locked group of other BASIC programs using the READU, READVU, and MATREADU statements.

2. Update by any other program of any item in the locked group.

3. Access of the group by the FILE-SAVE process.

The group will become unlocked when any item in that group is updated via a WRITE statement by the process which has it locked, when the BASIC program is terminated, or a RELEASE or DELETE statement unlocks the group. Items can be updated to the group without unlocking it by the program that issued the lock via the WRITEU, WRITEVU or MATWRITEU statements.

Other processes (such as those described in steps 1 through 3 above) which encounter a group lock will be suspended until the group becomes unlocked.

The maximum number of groups which may be locked by all processes in the system is 62. If a process attempts to lock a group when 62 locks are already set, it will be suspended until some group is unlocked.

Examples of the use of these statements:

Correct Use	Explanation
READU ITEM FROM INV, S5 THEN GOSUB	Locks group of items containing item S5. Reads S5 to variable ITEM; if S5 is read, executes the THEN clause. The group remains locked until one of its items is updated, or a RELEASE statement unlocks the group.
READVU ATT FROM B, "REC", 6 ELSE STOP	Locks group of items containing item REC. Reads attribute 6 to variable ATT or, if REC is non-existent, executes the ELSE clause. The group remains locked as in the preceding explanation.
MATREADU T FROM XM, "N4" ELSE NULL	This example shows use of a null ELSE clause to lock the group regardless of whether the item is existent or not.

Incorrect Use	Explanation
READU ELSE STOP	Parameters between "READU" and "ELSE" missing.
READVU X FROM F10, "GOR", 14	THEN/ELSE clause is missing.
MATREADU M, N FROM "BOY" ELSE STOP	Invalid parameter; only one array name may follow "MATREADU".

7.7.3 WRITE WITH LOCK FOR UPDATING: WRITEU, WRITEVU AND MATWRITEU STATEMENTS

The WRITEU, WRITEVU, and MATWRITEU statements have the letter "U" appended to them to imply update. These commands will not unlock the group locked by the program, but they will update this group when used by the program that issued the lock. The WRITEU, WRITEVU, MATWRITEU statements have the following general form:

```
WRITEU expression ON {file-variable,} item-name
WRITEVU expression ON {file-variable,} item-name, attribute-number
MATWRITEU array-variable ON {file-variable,} item-name
```

These commands are used primarily for master file updates when several transactions are being processed and an update of the master item is made following each transaction update.

If the group is not locked when the WRITEU, WRITEVU, or MATWRITEU statement is executed, the group will not be locked by the execution of the command.

7.7.3.1 RELEASE Statement

The RELEASE statement unlocks specified groups or all groups locked by the program.

The general form of the RELEASE statement is:

```
RELEASE ({file-variable,} item-name}
```

The RELEASE statement unlocks the group of the item-name specified. If the file-variable is used, the file will be the one previously assigned to that file-variable via an OPEN statement. If the file-variable is omitted, then the internal default variable is used (thus specifying the file most recently opened without a file-variable).

If the RELEASE statement is used without a file-variable or item-name, all groups which have been locked by the program will be unlocked. This form is:

```
RELEASE
```

The RELEASE statement is useful when an abnormal condition is encountered during multiple file updates. A typical programming sequence marks the item with an abnormal status, updates it to the file and then RELEASEs all other locked groups.

Examples of the use of these statements:

Correct Use	Explanation
WRITEU CUST.NAME ON CUST.FILE,ID	Replaces the current contents of the item specified by item-name ID (in the file opened and assigned to file-variable CUST.FILE) with the contents of CUST.NAME. Does not unlock the group.
WRITEVU CUST.NAME ON CUST.FILE,ID,3	Replaces the third attribute of item ID (in the file opened and assigned to file-variable CUST.FILE) with the contents of CUST.NAME. Does not unlock the group.
MATWRITEU ARRAY ON FILE.NAME,ID	Replaces the attributes of the item specified by ID (in the file opened and assigned to file-variable FILE.NAME) with the consecutive elements of vector ARRAY. Does not unlock the group.
RELEASE	Releases all groups locked by the program.
RELEASE CUST.FILE, PART.NO	Releases group containing item-name PART.NO in file CUST.FILE.

Incorrect Use	Explanation
WRITEVU LOC ON INC.FILE,PART.NO	Expression denoting attribute number is missing.
MATWRITEU ARRAY(5) ON ID	MATWRITEU takes entire array as a parameter. Subscript is not allowed.
RELEASE CUST.FILE	Item-name parameter is missing.

7.8 PROC I/O

7.8.1 PROCREAD STATEMENT

The PROCREAD statement will read the entire contents of the Primary Input Buffer of the controlling (currently running) PROC. The general form of the PROCREAD statement is:

```
PROCREAD variable THEN/ELSE statements
```

The contents of the Primary Input Buffer will be assigned to the specified variable. If the PROC is able to be read, the statement(s) following the THEN will be executed; if the PROC cannot be read, the statements following the ELSE will be executed. (The THEN/ELSE clause follows the same format as the THEN/ELSE clause in the IF statement.)

7.8.2 PROCWRITE STATEMENT

The PROCWRITE statement will replace the current contents of the Primary Input Buffer of the controlling (currently running) PROC with whatever the user specifies. The general form of the PROCWRITE statement is:

```
PROCWRITE expression
```

where expression is whatever you wish to place in the Primary Input Buffer.

7.9 TAPE I/O

BASIC programs may specify magnetic tape or cartridge disk I/O operations through the use of the READT (Read Tape Record), WRITET (Write Tape Record), WEOF (Write End-of-File Mark), and REWIND (Rewind Tape Unit) statements. The record length on the tape is the length specified in the most recent T-ATT statement executed at the TCL level.

7.9.1 READT STATEMENT

The READT statement reads the next record from the tape or cartridge disk unit. Its general form is:

```
READT variable THEN/ELSE statements
```

The next record is read and its string value is assigned to the variable indicated. Depending on whether the unit has been attached, or an End-of-File (EOF) mark is read, then the statement or sequence of statements following the THEN/ELSE will be executed. For example:

```
READT X ELSE PRINT "CANNOT READ"; STOP
```

or

```
READT X ELSE
    PRINT "END OF TAPE OR TAPE NOT ATTACHED"
    STOP
END
```

Here the next tape or cartridge disk record is read and assigned to variable X. If an EOF is read (or no unit is attached) then "CANNOT READ" or "END OF TAPE OR TAPE NOT ATTACHED" is printed and the program executes.

7.9.2 WRITET STATEMENT

The WRITET statement writes a record onto tape or cartridge disk. Its general form:

```
WRITET expression THEN/ELSE statements
```

The string value of the expression is written onto the next record of the tape or cartridge disk. Depending on whether the unit has been attached, or the string value of the expression is the empty string (''), the appropriate statement(s) following the THEN/ELSE will be executed. For example:

```
WRITET A ELSE STOP
```

This statement writes the string value of A onto the tape or cartridge disk. The program terminates if A='' or if no unit is attached.

7.9.3 WEOF AND REWIND STATEMENTS

The WEOF statement writes two EOF marks on the tape or cartridge disk, then backspaces over the second. This correctly positions the tape or disk for subsequent WRITET operations. The REWIND statement rewinds the tape unit to the Beginning-of-Tape (BOT). On the cartridge disk, the head is positioned to the first available sector. These statements have the following forms:

```
WEOF THEN/ELSE statements
REWIND THEN/ELSE statements
```

Depending on whether the tape or cartridge disk unit has been attached, the statement(s) following the THEN/ELSE will be executed. Examples of the use of Tape and Cartridge Disk I/O statements:

Correct Use	Explanation
READT B THEN PRINT "YES" GOTO 5 END	The next tape or cartridge disk record is read and its value assigned to variable B. If the tape or cartridge disk unit is attached and EOF is not read, then "YES" is printed and control passes to statement 5.
FOR I=1 TO 5 WRITET A(I) ELSE STOP NEXT I	The values of array elements A(1) through A(5) are written onto 5 tape or cartridge disk records. If one of the array elements has a value of '' (or if tape or cartridge disk unit not attached), the program will terminate.
WEOF ELSE STOP	Writes two EOF marks, then backspaces over the second one.
REWIND ELSE STOP	Tape is rewound to BOT (cartridge disk head is positioned to BOT).

Incorrect Use	Explanation
READT B+1 ELSE STOP	"B+1" is not a variable name.
WEOF	THEN/ELSE clause is missing.
REWIND	THEN/ELSE clause is missing.

7.10 STRING HANDLING

7.10.1 STRING SEARCHING: FIELD, COL1, AND COL2 FUNCTIONS

The FIELD function returns a substring from a string by specifying a delimiter character. The COL1 and COL2 functions return the numeric values of the column positions immediately preceding and immediately following the substring selected by the FIELD function.

7.10.1.1 FIELD

The general form of the FIELD function is:

```
FIELD("string","delimiter",occurrence#)
```

The FIELD function takes a string and searches for a substring delimited by the character specified in delimiter. Occurrence# specifies which occurrence of the substring is to be returned. If the occurrence is 1, then the FIELD function will return the substring from the beginning of the string up to the first occurrence of the delimiter. For example, the statement below assigns the string value of "XXX" to the variable A:

```
A = FIELD("XXX.YYY.ZZZ.555",".",1)
```

If occurrence# is 2, then the substring delimited by the first and second occurrence of the specified delimiter character will be returned. A value of 3 for occurrence# will return the substring delimited by the second and third occurrence of the specified delimiter character, and so on for higher values of the expression. For example, the statement below assigns the string value "ZZZ" to variable C:

```
C = FIELD("XXX.YYY.ZZZ.555",".",3)
```

7.10.1.2 COL1, COL2

The COL1 and COL2 functions have the following general form:

```
COL1()
COL2()
```

COL1() returns the numeric value of the column position immediately preceding the substring selected via the most recent FIELD function. For example:

```
B = FIELD("XXX.YYY.ZZZ.555",".",2)
BEFORE = COL1()
```

These statements assign the numeric value 4 to the variable BEFORE (the value "YYY" which is returned by the FIELD function is preceded in the original string by column position 4). COL2() returns the numeric value of the column position immediately following the substring selected via the most recent FIELD function. COL2() returns zero if the substring is not found. For example:

```
B = FIELD("XXX.YYY.ZZZ.555",".",2)
AFTER = COL2()
```

These statements assign the numeric value 8 to the variable AFTER (the value "YYY" which is returned by the FIELD function is followed in the original string by column position 8). Examples of the use of string searching functions:

Correct Use	Explanation
`T = "12345A6789A98765A"` `G = FIELD(T,"A",1)`	Assigns the string value "12345" to variable G.
`T = "12345A6789A98765A"` `G = FIELD(T,"A",3)`	Assigns the string value "98765" to variable G.

`Q = FIELD("ABCBA","B",2)`
`R = COL1()`
`S = COL2()`
	Assigns the string value "C" to variable Q, the numeric value 2 to variable R, and the numeric value 4 to variable S.

`X = "77$ABC$XX"`
`Y = "$"`
`Z = "ABC"`
`IF FIELD(X,Y,2)= Z THEN STOP`
	The IF statement will cause the program to terminate (the value returned by the FIELD function is "ABC", which equals the value of Z, thus making the test condition true).

Incorrect Use	Explanation
`A = FIELD("ABCDE","C")`	Occurrence expression is missing.
`COL1() = FIELD("XYZ","Z",L)`	"COL1()" is an Intrinsic Function and may not appear on the left side of an Assignment statement.
`Z = COL2(A)`	A function parameter may not be used with the COL1 and COL2 statement.

7.10.2 SEARCHING FOR A SUBSTRING: INDEX FUNCTION

The INDEX function searches a string for the occurrence of a substring and returns the starting column position of that substring. The general form of the INDEX function is:

```
INDEX("string","substring",occurrence#)
```

The INDEX function takes the string value of the first expression string and searches for the substring specified by the second expression. The third expression specifies which occurrence of that substring is sought. The resulting numeric value of the INDEX function is the starting column position of the substring within the string. A value of 0 is returned if the substring is not found. Consider the following example:

```
START = INDEX("ABCDEFGHI","DEF",1)
```

This statement assigns the value of 4 to the variable START (the first occurrence of the substring "DEF" starts at column position 4 of the string "ABCDEFGHI"). Next consider the example:

```
A = INDEX("AAXXAAXXAA","XX",2)
```

This statement assigns the value of 7 to variable A (the second occurrence of substring "XX" starts at column position 7 of string "AAXXAAXXAA").

The following example assigns a value of 0 to the variable VAR because the substring "Z" is not present within the string "ABC123":

```
Q = "Z"
R = "ABC123""
VAR = INDEX(R,Q,1)
```

Note that no blank space may appear between the function word "INDEX" and "(". This rule is true for all BASIC Intrinsic Functions.

Examples of the use of INDEX:

Correct Use	Explanation
`A = INDEX("ABCAB","A",2)`	Assigns value of 4 to variable A (2nd occurrence of "A" is at column position 4 of "ABCAB").

`X = 1`
`X = "1234ABC"`
`Y = "ABC"`
`IF INDEX(X,Y,1)=5 THEN GOTO 3`
	The IF statement will transfer control to statement 3 ("ABC" starts at column position 5 of "1234ABC" which makes the test condition in the IF statement TRUE).

`Q = INDEX("PROGRAM","S",5)`	Assigns value of 0 to variable Q ("S" does not occur in "PROGRAM").

`S = "X1XX1XX1XX"`
`FOR I=1 TO INDEX(S,"1",3)`
`.`
`.`
`NEXT I`
	The loop will execute 8 times (3rd occurrence of "1" appears at column position 8 of the string named S).

Incorrect Use	Explanation
`B = INDEX (B CAT Z, "XX",2)`	No space allowed between "INDEX" and "(".
`B = INDEX("QRS","S")`	Third expression is missing.
`INDEX("ZZZ33Q","33",1)`	An intrinsic function must appear as an expression or part of an expression; it may not be used as a stand-alone BASIC statement.

7.10.3 COUNTING OCCURRENCES OF A SUBSTRING: COUNT FUNCTION

The general form of the COUNT function is:

```
COUNT(´string´,´substring´)
```

The COUNT function counts the number of occurrences of a substring within a string. Any number of characters may be present in the substring. This function is particularly useful for determining the number of attributes within an item, or the number of multiple values or subvalues within an attribute.

The COUNT function returns a value of zero if the substring is not found, and returns the number of characters in the string if the substring is specified null. A null matches on any character. For example:

Command	Value of X
X = COUNT(´THIS IS A TEST´,´IS´)	2
X = COUNT(´THIS IS A TEST´,´X´)	0
X = COUNT(´THIS IS A TEST´,´´)	14

There are 14 characters in the string ´THIS IS A TEST´. Another example:

Command	Value of X
X = COUNT(´AAAA´,´AA´)	3

There are 3 substrings within the string AAAA.

```
AAAA          STRING
AA            SUBSTRING 1
 AA           SUBSTRING 2
  AA          SUBSTRING 3
```

7.10.4 COUNTING DELIMITED VALUES: DCOUNT FUNCTION

The general form of the DCOUNT function is:

```
DCOUNT('string','delimiter')
```

The DCOUNT function counts the number of values separated by a specified delimiter. The DCOUNT function differs from the COUNT function in that it returns the true number of values by the specified delimiter, rather than the number of occurrences of the delimiter within the string. For example:

```
AM = CHAR(254)
A = "ABC":AM:"DEF":AM:"GHI":AM:"JKL"
```

Command	Value of X
X = COUNT(A,AM)	3
X = DCOUNT(A,AM)	4

The DCOUNT function may be used to count the number of attributes in an item, or the number of values (or subvalues) within an attribute. The DCOUNT function returns a value of zero when a null string is encountered.

Examples of the use of COUNT, DCOUNT:

Correct Use	Explanation
A = "1234ABC5723" X = COUNT(A,'23')	Value returned in X is 2. (There are two occurrences of '23' in the string A.)
X = COUNT('ABCDEFG','')	Value returned in X is 7. (A null substring will match any character.)
AM = CHAR(254) A = "123":AM:"456":AM:"ABC" X = DCOUNT(A,AM)	Value returned in X is 3. (There are three values in the string separated by attribute marks.)
VM = CHAR(253) A = "123]456":AM:"ABC]DEF]HIJ" X = DCOUNT(A,VM)	Values returned in X is 4. (There are four values in the string separated by value marks.)
A = "ABCDEFG" X = DCOUNT(A,'')	Value returned in X is 0. (A null is specified as the delimiter.)

Incorrect Use	Explanation
A = "THIS IS A TEST" X = COUNT(A,IS)	If a literal substring is used, it must be enclosed in single or double quotes.
X = DCOUNT(A)	Both COUNT and DCOUNT functions must have two expressions.

7.10.5 STRING SPACING: SPACE AND TRIM FUNCTIONS

The SPACE function generates a string value containing a specified number of blank spaces. The TRIM function removes extraneous blank spaces from a specified string.

7.10.5.1 SPACE

The general form of the SPACE function is:

```
SPACE(expression)
```

The SPACE function generates a string value containing the number of blank spaces specified by the expression. For example:

```
PRINT SPACE(10):"HELLO"
```

This statement prints 10 blanks followed by the string "HELLO".

7.10.5.2 TRIM

The general form of the TRIM function is:

```
TRIM(expression)
```

The TRIM function deletes preceding, trailing, and redundant blanks from the literal or variable expression. For example:

```
A='       GOOD MORNING,             MR. BRIGGS'
A=TRIM(A)
PRINT A
```

The PRINT statement will print:

```
GOOD MORNING, MR. BRIGGS
```

Examples of the use of SPACE, TRIM:

Correct Use	Explanation
B = 14 A = SPACE(B)	Assigns to variale A the string value containing 14 blank spaces.
DIM M(10) MAT M = SPACE(20)	Assigns a string consisting of 20 blanks to each of the 10 elements of array M.
S = SPACE(5) L = "SMITH" C = "," F = "JOHN" N = S:L:S:C:S:F	Assigns to variable N the concatenated string consisting of 5 blanks, the name SMITH, 5 blanks, a comma, 5 blanks, and the name JOHN.
M = TRIM(N)	Where N is the above string variable, assigns to variable M a string consisting of the name SMITH, 1 blank, a comma, 1 blank, and the name JOHN.

Incorrect Use	Explanation
Q = SPACE()	Expression is missing.
P = SPACE+A	Argument and parentheses are missing.
X = TRIM(X,Y)	Only one expression is allowed for TRIM function.
TRIM(A)	Function cannot stand alone; it must appear in a valid BASIC statement.

7.10.6 STRING REPETITION AND LENGTH DETERMINATION

7.10.6.1 STR

The STR function generates a string value containing a specified number of occurrences of a specified string. The general form of the STR function is:

```
STR(´string´,occurrence#)
```

where ´string´ specifies the string that is to be repeated the number of times specified by occurrence number. Note that string may also be a variable to which a string value has been assigned. The following statement, for example, assigns a string value containing 12 asterisk characters to variable X:

```
X=STR(´*´,12)
```

As a further example, the following statement will cause the string value "ABCABCABC" to be printed:

```
PRINT STR(´ABC´,3)
```

7.10.6.2 LEN

The LEN function gives the length of a string. The general form is:

```
LEN(expression)
```

The numeric value that gives the length of the string (number of bytes) specified by the expression will be returned. For example:

```
A = "1234ABC"
B = LEN(A)
```

These statements assign the value of 7 to variable B.

Examples of the use of STR, LEN:

Correct Use	Explanation
`VAR = STR("A",5)`	Assigns to variable VAR the string value containing five A's.
`Q = LEN("123")`	Assigns the value 3 to variable Q (the length of string "123").

`A = 'BBB'`
`B = STR("B",3)`
`C = B CAT A`
	Assigns to variable C the string value containing six B's.

`X = "123"`
`Y = "ABC"`
`Z = LEN(X CAT Y)`
	Assigns the value 6 to variable Z.

`N = STR("?%?",4)`	Assigns to variable N the string value containing 4 consecutive occurrences of the string "?%?".

Incorrect Use	Explanation
`S = STR(NAME,2)`	Quotes missing that should enclose string NAME, unless NAME is a variable.
`J = STR ("Z",40)`	No space allowed between "STR" and "(".
`W = LEN("Z",40)`	Only one expression allowed for LEN function.

7.11 DYNAMIC ARRAY OPERATIONS

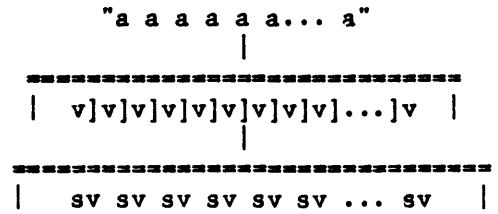
7.11.1 DYNAMIC ARRAY STRUCTURE

BASIC contains a number of statements and functions that are extremely useful for accessing and updating PICK files. A brief description of the PICK file structure as it concerns the BASIC programmer is appropriate at this point.

A PICK file consists of a set of items. When a PICK file item is accessed by a BASIC program, it is represented as a BASIC string in item format. A string in item format is called a dynamic array.

A dynamic array consists of one or more attributes separated by attribute marks. An attribute mark has an ASCII equivalent of 254, which prints as "^". An attribute, in turn, may consist of a number of values separated by value marks. A value mark has an ASCII equivalent of 253, which prints as "]". Finally, a value may consists of a number of secondary values separated by secondary value marks. A secondary value mark has an ASCII equivalent of 252, which prints as "\".

The general form of a dynamic array is illustrated below:



where:
- a = attribute
- v = value
- sv = secondary value

An example of a dynamic array is:

"55^ABCD^732XYZ^100000.33"

where "55", "ABCD", "732XYZ", and "100000.33" are attributes.

The following illustrates a more complex dynamic array:

"Q5^AAAA^952]ABC]12345^A^B^C]TEST\12I\9\99.3]2^555"

where "Q5". "AAAA", "952]ABC]12345", "A", "B", "C]TEST\12I\9\99.3]2" and "555" are attributes; "952", "ABC", 12345", "C", "TEST\12I\9\99.3", and "2" are values; and "TEST", "12I", "9", and "99.3" are secondary values.

Dynamic arrays can be directly manipulated by the BASIC dynamic array functions. Dynamic arrays are called "arrays" because they can be referenced by these functions using 3 subscripts. They are "dynamic" in the sense that elements can be added and deleted without having to recompile the program, as long as the item does not exceed 32,267 characters.

Further examples of correctly formed dynamic arrays:

Array	Explanation
123^456^789]ABC]DEF	"123", "456", "789]ABC]DEF" are attributes; "789", "ABC" and "DEF" are values.
1234567890	"1234567890" is an attribute.
Q56^3.22]3.56\88\B]C^99	"Q56", "3.22]3.56\88\B]C", and "99" are attributes; "3.22", "3.56\88\B", and "C" are values; "3.56", "88", and "B" are secondary values.
A]B]C]D^E]F]G]H^I]J	"A]B]C]D", "E]F]F]H", and "I]J" are attributes; "A", "B", "C", "D", "E", "F", "G", "H", "I", and "J" are values.

7.11.2 LOCATE STATEMENT

The LOCATE statement may be used to find the index of an attribute, a value, or a secondary value within a dynamic array. The elements of the dynamic array may be specified in ascending or descending ASCII sequence, and sorted with either right or left justification. If the specified attribute, value, or secondary value is not present in the dynamic array in the proper sequence, an index value is returned which may be used in an INSERT statement to place the sought element into its proper location. The general form of the LOCATE statement is:

```
LOCATE( 'String', Item {, Attr# {, Val#}}; Var {; 'Seq'} )
THEN/ELSE statement
```

"String" is the element to be located in dynamic array "item". "Var" is the variable into which the index of "String" is to be stored. "Attr#" and "Val#" are optional parameters which restrict the scope of the search within "Item". If neither parameter is present, "String" is tested for equality with attributes in "Item", and "Var" returns an attribute number. If "Attr#" is present, "String" is compared with values within the attribute specified by "Attr#" of "Item", and "Var" returns a value number. If "Val#" is also present, the search is conducted for secondary values of the specified attribute and value of "Item", and "Var" returns a secondary value number.

If "Seq" has the value "A" (or any string value beginning with "A"), the elements of "Item" are assumed to be sorted in ascending sequence. If "Seq" has the value "D" (or any string value beginning with "D"), the elements are assumed to be in descending sequence. All other values for "Seq" are ignored.

If the first character of "Seq" is either "A" or "D", the second character determines the justification used when sorting the elements. If this character is "R", right justification is used. (This is useful with numeric elements.) If this value is "L" (or any other value, including null), left justification is used.

LOCATE statements may be used to locate and/or insert controlling and dependent associative attributes within dictionary items. The following example demonstrates how a file may be searched for items which contain 'dependent' associative attributes in attribute 4 and how, if the 'D' is not located, it will be inserted by the execution of the 'ELSE' clause:

```
LOCATE('D',ITEM,4;VAR) ELSE ITEM = INSERT(ITEM,4,VAR,0,'D')
```

This single statement eliminates the need for a loop which would have to specifically extract and test the attribute and provide two consequent paths before the next item could be searched.

The THEN/ELSE clause of the LOCATE statement operates exactly like the THEN/ELSE clause of the IF statement.

Examples of the use of LOCATE statements:

Correct Use	Explanation
LOCATE(´55´,ITEM,3,1;VAR;´AR´) ELSE ITEM = INSERT(ITEM,3,1,VAR,´55´) END	The third attribute, first value of array ´ITEM´ is searched for the numeric literal ´55´. ´VAR´ will return with the secondary value index if the numeric is found, and will return with the correct secondary value index if the numeric is not found. If it is not found, control passes to the ELSE clause which inserts the numeric into the correct position by virtue of the index contained in ´VAR´. The optional parameter ´AR´ specifies ascending sequence and right justification.

Incorrect Use	Explanation
LOCATE(123,ITEM,2;VAR;AR)	String 123 and AR must be surrounded by quotes. Also the THEN/ELSE clause is missing.
LOCATE(´123´,ITEM,VAR,´AR´) ELSE STOP	A semicolon must precede VAR and Seq; commas are used to separate STRING, ITEM, ATTR#, and VAL#.

7.11.3 EXTRACT FUNCTION

The EXTRACT function returns an attribute, a value, or a secondary value from a dynamic array. The general form of this function is:

```
value=da<exp{,exp}{,exp}>
```

The dynamic array used by this function is specified by da. Whether an attribute, a value, or a secondary value is extracted depends upon the values inside the '< >'. The first expression specifies an attribute, the second specifies a value, and the third specifies a secondary value. The second and third expressions are optional. Consider the following example:

```
OPEN 'TEST' TO TEST ELSE STOP 201, 'TEST'
READ ITEM FROM TEST, 'NAME' ELSE STOP 202, 'NAME'
PRINT ITEM <3,2>
```

These statements cause value 2 of attribute 3 of item ITEM in file TEST to be printed.

Another example:

```
OPEN 'ACCOUNT' TO ACCOUNT ELSE STOP 201, 'ACCOUNT'
READ ITEM1 FROM ACCOUNT, 'ITEM1' ELSE STOP 202, 'ITEM1'
IF ITEM1<3,2,1>=25 THEN CRT "MATCH"
```

These statements cause the message "MATCH" to be printed if secondary value 1 of value 2 of attribute 3 of item ITEM1 in file ACCOUNT is equal to 25.

Other EXTRACT function forms are:

```
EXTRACT(da,am,vm,svm)          da<am>
```

where:
- da is dynamic array
- am is attribute mark (count)
- vm is value mark (count)
- svm is secondary value mark (count)

Note that trailing subvalue or value mark counts are no longer required.

Examples of the use of EXTRACT:

Correct Use	Explanation
`Y=X<2>`	Assigns attribute 2 of dynamic array X to variable Y.

`A=3`
`B=2`
`Q1=ARR<A,B,A+1>`
	Assigns secondary value 4 of value 2 of attribute 3 of dynamic array ARR to variable Q1.

`IF B<3,2,1> > 5 THEN`
`  PRINT MSG`
`  GOSUB 100`
`END`
	If secondary value of 1 of value 2 of attribute 3 of dynamic array B is greater than 5, then the value of MSG is printed and a subroutine branch is made to statement 100.

`PRINT D<25,2>`	Prints value 2 of attribute 25 of dynamic array D.

Incorrect Use	Explanation
`ITEM=DA<3`	There is no closing '>' present.
`ITEM=<3,2>`	Dynamic array name is missing.

7.11.4 REPLACE FUNCTION

The REPLACE function replaces an attribute, a value, or a secondary value in a dynamic array. The general form of REPLACE function is:

```
    da<exp{,exp}{,exp}>=VALUE
or
    REPLACE(da,exp{,exp}{,exp};exp)
```

The dynamic array used by this function is specified by da. Whether an attribute, a value, or a secondary value is replaced depends upon the values of the first, second and third expressions. The first expression specifies an attribute, the second specifies a value, and the third specifies a secondary value. In the case of the second usage, the fourth expression is the new data. The second and third expressions are optional in both cases.

The following example replaces attribute 4 of item NAME in file INVENTORY with the string value "EXAMPLE":

```
    OPEN 'INVENTORY' TO INV ELSE STOP 201, 'INVENTORY'
    READ NAME FROM INV, 'NAME' ELSE STOP 202, 'NAME'
    NAME<4>='EXAMPLE'
    WRITE NAME ON INV, 'NAME'
```

Alternatively, this operation could have been written as follows:

```
    OPEN 'INVENTORY' TO INV ELSE STOP 201,'INV'
    READ NAME FROM INV,'NAME' ELSE STOP 202, 'NAME'
    WRITE REPLACE(NAME,4;'EXAMPLE') ON INV, 'NAME'
```

If the second, third, or fourth expression of the REPLACE function has a value of -1, then insertion after the last attribute, last value, or last secondary value (respectively) of the dynamic array is specified. For example:

```
    OPEN 'XYZ' TO XYZ ELSE STOP 201, 'XYZ'
    READ B FROM XYZ 'ABC' ELSE STOP 202, 'ABC'
    B<3,-1>='NEW VALUE'
    WRITE B ON XYZ,'ABC'
```

These statements insert the string value "NEW VALUE" after the last value of attribute 3 of item ABC in file XYZ. Other REPLACE function forms are:

```
    REPLACE(da,am,vm,svm)    da<am>
```

where:

da	is	dynamic array
am	is	attribute mark (count)
vm	is	value mark (count)
svm	is	secondary value mark (count)

Examples of the use of REPLACE:

Correct Use	Explanation
X<4>=‘’	Replaces attribute 4 of dynamic array X with the empty (null) string.
Y=REPLACE(X,4;‘’)	Replaces attribute 4 of dynamic array X with the empty (null) string, and assigns the resultant array to Y.
VALUE="TEST STRING" DA<4,3,2>=VALUE	Replaces secondary value 2 of value 3 of attribute 4 in dynamic array DA with the string value "TEST STRING".
X="ABC123" Y<1,1,-1>=X	Inserts the value "ABC123" after the last secondary value of value 1 of attribute 1 in dynamic array Y.
A=REPLACE(B,2,3;"XXX")	Replaces value 3 of attribute 2 of dynamic array B with the value "XXX", and assigns the resultant dynamic array to A.

Incorrect Use	Explanation
B<1,2,3=VALUE	Closing ‘>’ is missing.
V5=REPLACE(V4,4,0,0,‘TEST’	Terminating parentheses is missing.
REPLACE(X,3,3,3,‘ABC’)	REPLACE is a function and may therefore not appear as a stand alone statement.

7.11.5 INSERT FUNCTION

The INSERT function inserts an attribute, a value, or a secondary value into a a dynamic array. The general form of this function is:

```
INSERT(da,exp{,exp}{,exp};exp)
```

The dynamic array used by this function is specified by da. Whether an attribute, a value, or a secondary value is inserted depends upon the values of the first, second and third expressions. The first expression specifies an attribute, the second specifies a value, and the third specifies a secondary value. The second and third expressions are optional. The value to be inserted is specified by the fourth expression. Consider the following example:

```
OPEN 'TEST-FILE' TO TEST ELSE STOP 201, 'TEST-FILE'
READ X FROM TEST-FILE,'NAME' ELSE STOP 202,'NAME'
X = INSERT(X,10;'XXXXX')
WRITE X ON TEST,'NAME'
```

These statements insert the value "XXXXX" before attribute 10 of item NAME, thus creating a new attribute.

If the first, second or third expression of the INSERT function has a value of -1, then insertion after the last attribute, last value, or last secondary value (respectively) of the dynamic array is specified. For example:

```
OPEN 'FM1' TO FM1 ELSE STOP 201,'FM1'
READ B FROM FM1,'IT5' ELSE STOP 202,'IT5'
A = INSERT(B,-1;'EXAMPLE')
WRITE A ON FM1,'IT5'
```

These statements insert the string value "EXAMPLE" after the last attribute of item IT5 in file FM1.

Other INSERT function forms are:

```
INSERT(da,am,vm,svm,new)      INSERT(da,am;new)
```

where:

	da	is dynamic array
	am	is attribute mark (count)
	vm	is value mark (count)
	svm	is secondary value mark (count)
	new	is new data to be inserted

Note that trailing value and secondary value mark parameters are no longer required. However, if they are omitted when using the INSERT function, a semicolon must be used before the new data to be inserted.

Examples of the use of INSERT:

Correct Use	Explanation
`Y = INSERT(X,3,2;"XYZ")`	Inserts before value 2 of attribute 3 of dynamic array X the string value "XYZ" (thus creating a new value), and assigns the resulting dynamic array to variable Y.
`NEW = "VALUE"` `TEMP = INSERT(TEMP,9;NEW)`	Inserts before attribute 9 of dynamic array TEMP the string value "VALUE" (thus creating a new attribute).
`A = "123456789"` `B = INSERT(B,3,-1;A)`	Inserts the value "123456789" after the last value of attribute 3 of dynamic array B.
`Z = INSERT(W,5,1,1;"B")`	Inserts the string value "B" before secondary value 1 of value 1 of attribute 5 in dynamic array W (thus creating a new secondary value), and assigns the resulting dynamic array to variable Z.

Incorrect Use	Explanation
`B7 = INSERT (B7,1,1,1;"AA")`	A space must not appear between "INSERT" and "(".
`A = INSERT(B,4,1,1,1;"XYZ")`	Too many expressions.
`B = INSRT(B,4,1,1,"XYZ")`	INSERT is spelled incorrectly.

7.11.6 DELETE FUNCTION

The DELETE function deletes an attribute, a value, or a secondary value from a dynamic array. The general form of the DELETE function is:

```
DELETE(da,exp{,exp}{,exp})
```

The dynamic array used by this function is specified by da. Whether an attribute, a value, or a secondary value is deleted depends upon the values of the first, second and third expressions. The first expression specifies an attribute, the second specifies a value, and the third specifies a secondary value. The second and third expressions are optional.

If a value is deleted (i,e., the secondary value expression is zero), the value mark associated with the value is also deleted. If an attribute is deleted (i.e., the value and the secondary value expression are both zero), the attribute mark associated with the attribute is also deleted. Consider the following example:

```
OPEN 'INVENTORY' TO INV ELSE STOP 201,'INVENTORY'
READ VALUE FROM INV, 'ITEM2' ELSE STOP 202, 'ITEM2'
VALUE = DELETE(VALUE,1,2,3)
WRITE VALUE ON INV,'ITEM2'
```

These statements delete secondary value 3 of value 2 of attribute 1 of item ITEM2 in file INVENTORY. The delimiter associated with secondary value 3 is also deleted. Consider the following example:

```
OPEN 'TEST' TO TEST ELSE STOP 201,'TEST'
READ X FROM TEST,'NAME' ELSE STOP 202,'NAME'
WRITE DELETE(X,2) ON TEST,'NAME'
```

These statements delete attribute 2 (and its associated delimiter) of item NAME in file TEST.

Other forms of the DELETE function are:

```
DELETE(da,am,vm,svm)    DELETE(da,am)
```

where:

da	is	dynamic array
am	is	attribute mark (count)
vm	is	value mark (count)
svm	is	secondary value mark (count)

Examples of the use of DELETE:

Correct Use	Explanation
`Y = DELETE(X,3,2)`	Deletes value 2 of attribute 3 of dynamic array X (and its associated delimiter), and assigns resulting dynamic array to Y.
`A=1;B=2;C=3` `DA = DELETE(DA,A,B,C-A)`	Deletes secondary value 2 (and its associated delimiter) of value 2 of attribute 1 of dynamic array DA.
`X = DELETE(X,7)`	Deletes attribute 7 (and its associated delimiter) of dynamic array X.
`PRINT DELETE(X,7,1)`	Prints the dynamic array which results when value 1 of attribute 7 of dynamic array X is deleted.

Incorrect Use	Explanation
`B = DELETE(CM5,"XYZ")`	Strings such as "XYZ" are not allowed in a DELETE function.
`DELETE = A(5,5)`	DELETE must not be used as a variable name.

terminal and printer input and output 8

8.1 TERMINAL INPUT

8.1.1 INPUT AND PROMPT STATEMENTS

The INPUT statement is used to request input data from the user's terminal. The PROMPT statement is used to select the "prompt character" which is printed at the terminal to prompt the user for input. The general form of the INPUT statement is:

```
INPUT variable {,n}{:}
```

Upon execution of an INPUT statement, a "prompt" character will be printed at the user's terminal. The user's response to the prompt will then be assigned to the variable indicated in the INPUT statement. For example:

```
INPUT A
```

This statement will cause a prompt character to be printed at the user's terminal. The data which the user thereupon inputs will become the current value of variable A. If n is used, input is only accepted if less than or equal to n characters. When the nth character is input, there will be an automatic carriage return and line feed.

A colon may be used following the input variable to suppress the automatic carriage return and line feed that occur when a value is input. For example:

```
INPUT AMOUNT:
```

Via the PROMPT statement, the user may select any character to be used as the input prompt character. The PROMPT statement has the following general form:

```
PROMPT expression
```

The expression used becomes the prompt character. For example:

```
PROMPT ":"
```

This statement selects the character ":" as the prompt character for subsequent INPUT statements.

If the value of the expression is a numeric value of more than 1 digit, or a string consisting of more than one character, only the most significant character will be used. For example:

```
PROMPT "ABC"
```

This statement selects the character "A" as a prompt character. (Quotes are needed if a string expression is used.) When a PROMPT statement has been executed, it will remain in effect until another PROMPT statement is executed. If a PROMPT statement has not been executed, the INPUT statement will use a question mark (?) as the prompt character (i.e., "?" is the default prompt character).

Examples of the use of these statements:

Correct Use	Explanation
INPUT VAR	Will request a value for variable VAR at the user's terminal.
PROMPT "@"	Specifies that the character @ will be used as a prompt character for subsequent INPUT statements.
PROMPT A	Specifies that the current value of A will be used as a prompt character.

Incorrect Use	Explanation
INPUT	Variable is missing.
INPUT X,Y	Only one input may be specified per INPUT statement.
INPUT "STRING"	"STRING" is not a variable name.
PROMPT	Expression is missing.
PROMPT *	The character * must be enclosed in quotes.

8.1.2 MASKED INPUT

The general form of the Masked Input statement:

```
INPUT @(x,y):variable mask
```

This is a very complex input function. It is capable of replacing as many as 20 lines of BASIC code used in screen input. Its functions include cursor addressing, output masking, editing, error messages, input masking, and exception trapping.

This command itself is used for the actual entry of the data. Ancillary functions can be performed by the commands described below. In the general form above, "variable" represents the name of the variable being input, and "mask" represents a standard PICK format mask. If the variable being used already has a value, it will be displayed at the specified cursor address using "mask" as the output mask. Regardless, the cursor is positioned one character back of "x" in the "@(x,y)" specification, the prompt character is printed and input is requested. If the user presses the return key, then whatever default value was there before will be accepted. Otherwise, the input will be verified against the mask, and if acceptable, will be assigned to "variable". If the mask contains a decimal digit specification and/or a scaling factor, then numeric checking will be performed. If the mask contains a length specification (for example, R#10), then length checking will be performed. If the mask is 'D' (or any other valid date mask), then a date verification will be performed.

Note that data is converted on output and input. Thus, if you wish to input a date, the default should be stored in internal format. It will be displayed and input in output format and will be placed back in the variable in internal format. Note also that the '%' is a numeric character verification symbol. Thus, if the statement executed is INPUT @(20,10):SOC.SEC '%%%-%%-%%%%' and the data entered is 423-15-6897 then SOC.SEC will contain the value 423156897. If an error condition is encountered, then a message is printed at the bottom of the screen. Examples of Masked Input statements:

`INPUT @(25,2):INV.DATE 'D'`	Inputs a date.
`INPUT @(35,7):AMOUNT 'R2,'`	Inputs a dollar value.
`INOUT @(20,14):NAME 'L#40'`	Inputs a test field with a length specification.
`INPUT @(0,10):DESC`	Inputs data with no mask.

8.1.3 OTHER INPUT FORMS

```
INPUTERR expr

INPUTTRAP 'character string' GOTO n,n,n,n ...
     or
INPUTTRAP 'character string' GOSUB n,n,n,n ...

INPUTNULL 'x'
```

These are all support functions for the INPUT statement. They allow the user to tailor the INPUT function to conform to local standards.

INPUTERR causes a message, specified by "expr", to be printed on the last line of the screen. This differs from an explicit PRINT statement in that it sets a flag indicating that a message has been printed. Thus, when the next valid entry is made, the system will check the flag and clear the bottom line.

INPUTTRAP allows the user to set a trap for a particular character or characters. Each character in the string specification corresponds to a label in the GOTO or GOSUB clause. Thus, for example, if the statement INPUTTRAP '_X' GOTO 10,20 is executed, the subsequent entry of a '_' character will cause a branch to "10" and the entry of 'X' will cause a branch to "20". The GOSUB form of this expression will cause a subroutine call to be issued instead. CAUTION: The subroutine RETURN statement will cause a return to the statement following the INPUTTRAP statement; not the one following the INPUT statement.

The INPUTNULL statement allows the user to define a character which is to signify that whatever default value was present is to be replaced by the null string. (This feature applies to masked input statements only.) Thus, if the statement INPUTNULL '/' is executed, the subsequent entry of a '/' character will cause a defaulted value to go to null. Note that the default character is '_'. Examples of these INPUT forms:

`INPUTERR 'INVALID DATA!'`	Displays error message.
`INPUTTRAP '*/' GOTO 150,170`	Causes branching if either '*' or '/' is entered.
`INPUTNULL '@'`	Causes the '@' character to null defaults in masked INPUT statements.

8.2 SYSTEM OUTPUT: DEVICE SELECTION

8.2.1 PRINTER STATEMENTS

The PRINTER statement selects either the user's terminal or the line printer for subsequent program output. The PRINTER statement takes on three forms:

```
PRINTER ON
PRINTER OFF
PRINTER CLOSE
```

The PRINTER ON statement directs program output data specified by subsequent PRINT, HEADING, or PAGE statements to be output to the line printer.

The PRINTER OFF statement directs subsequent program output to the user's terminal.

Once executed, a PRINTER ON or PRINTER OFF statement will remain in effect until a new PRINTER ON or PRINTER OFF statement is executed. If a PRINTER ON statement has not been executed, all output will be to the user's terminal.

When a PRINTER ON statement has been issued, subsequent output data (specified by PRINT, HEADING, or PAGE statements) is not immediately printed on the line printer unless immediate printing is forced via the system SP-ASSIGN I option. (See the SPOOLER Manual.) Rather, the data is stored in an intermediate buffer area and is automatically printed upon termination of program execution.

The PRINTER CLOSE statement will cause all data currently stored in the intermediate buffer area to immediately be printed. This is used then it is necessary that the data be printed on the line printer prior to program termination.

When a PRINTER OFF statement has been issued, subsequent output data is always printed at the user's terminal immediately upon execution of the PRINT, HEADING, or PAGE statements. The PRINTER CLOSE statement applies only to output data directed to the line printer.

Examples of the use of PRINTER statements:

Correct Use	Explanation
PRINTER ON PRINT A PRINTER OFF PRINT B	Causes the value of variable B to be immediately printed at the user's terminal, and the value of variable A to be printed on the line printer when the program is finished executing.
PRINTER ON PRINT A PRINTER CLOSE PRINTER OFF PRINT B	Causes the value of variable A to be immediately printed on the line printer, and thereafter causes the value of variable B to be printed at the user's terminal.
PRINTER ON PRINT A PRINTER OFF PRINT B PRINTER CLOSE	Causes the value of variable B to be immediately printed at the user's terminal, and thereafter causes the value of variable A to be printed on the line printer.

Incorrect Use	Explanation
PRINTER	"ON", "OFF", or "CLOSE", is missing.
PRINTER =500	"PRINTER" may not be used as a variable name.
PRINTER X	Illegal format.

8.2.2 PRINT STATEMENT

The PRINT statement outputs data to the device selected by the PRINTER statement. The PRINT ON option allows output to multiple print files. The general form of the PRINT statement is:

```
PRINT {ON expression} print-list
```

The PRINT statement without the ON option is used to output variable or literal values to the terminal or line printer, as previously selected by a PRINTER statement. The print-list may consist of a single expression, or a series of expressions, separated by commas or colons (these punctuation marks are used to denote output formatting; refer to Tabulation and Concatenation in PRINT Statement, which is discussed in the next subsection). An expression may be any legal BASIC expression. The following statement, for example, will print the current value of the expression X+Y:

```
PRINT X+Y
```

The PRINT ON expression version of the PRINT statement is used when PRINTER ON is in effect to output the print-list items to a numbered print file. This is usually done when building several reports at the same time, each having a different number. The expression following ON indicates the print file number, which may be from 0 to 254 (selected arbitrarily by the print file program). Consider the following example:

```
PRINT ON 1 A,B,C,D
PRINT ON 2 E,F,G,H
PRINT ON 3 X,Y,Z
```

These statements will generate 3 separate output listings, one containing A, B, C, and D values, one containing E, F, G and H values, and the third ' containing X, Y and Z values.

When the ON expression is omitted, print file zero is used.

The HEADING statement affects only print file zero. Pagination must be handled by the program for print files other than zero. Lack of pagination will result in continuous printing across page boundaries.

When PRINTER OFF is in effect, both PRINT ON and PRINT operate identically (i.e., all output is to the terminal). The contents of all print files used by the program, including print file zero, will be output to the printer in sequence when a PRINTER CLOSE statement is given or on termination of the program.

Examples of the use of PRINT statements:

Correct Use	Explanation
PRINTER ON PRINT X	Causes the value of X to be output to print file 0.
PRINTER ON PRINT ON 24 X	Causes the value of X to be output to print file 24.
N=50 PRINT ON N X,Y,Z	Outputs print-file to print file 50.
PRINTER ON PRINT ON 15 "100" PRINT ON 40 "100"	Causes the value 100 to be copied to both print file 15 and print file 40.
PRINTER ON PRINT A PRINT B	Print file 0 will contain the values of A and B.
PRINTER ON PRINT ON 10 F1,F2,F3 PRINT ON 20 M,N,P PRINT ON 10 F4,F5,F6	Print file 10 will contain the values of F1 through F6; print file 20 will contain the values of M, N, and P.

Incorrect Use	Explanation
PRINT, C	Extraneous comma.
PRINT ON A, B	Missing print file number.
PRINT ON 300	Invalid print file number.

8.2.2.1 Tabulation and Concatenation in PRINT Statement

The print-list of the PRINT statement may specify tabulation or concatenation when printing multiple items.

Output values may be aligned at tab positions across the output page by using commas to separate the print-list expressions. Tab positions are preset at every 18 character positions. Consider the following example:

```
PRINT (50*3)+2, A, "END"
```

Assuming that the current value of A is 37, this statement will print the values across the output page as follows:

```
152          37           END
```

Output values may be printed continuously across the output page by using colons to separate the print-list expressions. The following statement, for example, will cause the text message "THE VALUE OF A IS 5010" to be printed:

```
PRINT "THE VALUE OF A IS ":50:5+5
```

After the entire print-list has been printed, a carriage return and a line feed will be executed, unless the print-list ends with a colon. In that case, the next value in the next PRINT statement will be printed on the same line as the very next character position. For example, these statements:

```
PRINT A:B,C,D:
PRINT E,F,G
```

will produce exactly the same output as this statement:

```
PRINT A:B,C,D:E,F,G
```

Examples of the use of the PRINT statement print-list formatting:

Correct Use	Explanation
PRINT A:B: PRINT C:D: PRINT E:F	Prints the current values of A, B, C, D, E, and F contiguously across the output page, each value concatenated to the next.
PRINT A=1	Prints 1 if "A=1" is true; prints 0 otherwise.
PRINT A*100,Z	Prints the value of A*100 starting at column position 1; prints the value of Z on the same line starting at column position 18 (i.e., 1st tab position).
PRINT	Prints an empty (blank) line.
PRINT "INPUT":	Prints the text "INPUT" and does not execute a carriage return or line feed.
PRINT " ", B	Prints the value of B starting at column position 18 (i.e., 1st tab position).

Incorrect Use	Explanation
PRINT "RESULTS	Terminating quote mark is missing.
PRINT X, Y, Z,	Print-list must not terminate with a comma.

8.2.3 CRT STATEMENT

The CRT statement outputs data on the terminal. It does so regardless of whether a PRINTER ON statement is in effect or a 'P' option has been used with the RUN statement. The general form of the CRT statement is:

```
CRT print-list
```

The print-list may consist of a single expression, or a series of expressions, separated by commas or colons. The expressions may be any legal BASIC expression. The following statement, for example, will print the current value of the expression A+B-C on the terminal.

```
CRT A+B-C
```

The following expression will print the string in quotes on line 5 at column position 3 on the terminal:

```
CRT @(3,5):"PROGRAM C ENTERED"
```

8.3 OUTPUT FORMATTING

8.3.1 TERMINAL CURSOR CONTROL AND SCREEN FUNCTIONS: @

The @ function generates the terminal output codes required to position the cursor to a specified position. When used with the negative values specified below, the @ function generates cursor-control and screen formatting characters. The @ function takes on three forms:

```
@(column#)
@(column#,line#)
@(-n)
```

The first form generates the cursor address to position the cursor at the column position specified by column #. (The line position will be the current line.) The second form generates the cursor address to position the cursor at the column position specified by column #, and the line specified by line #. For example:

```
PRINT @(30): "HELLO"
```

This statement prints the message "HELLO" on the current line position of the cursor, starting at column position 30. Another example:

```
X = @(10,15):
PRINT X: "GOOD-BYE"
```

This statement prints the message "GOOD-BYE" on line 15, starting at column position 10. The values of the expression(s) used in the @ function must be within the row and column limits of the terminal screen.

When the @ function is used with a negative value specified by -n, it generates the special cursor-control characters for the current terminal type (as defined by the TERM statement in effect at the time). (Note that not every type of terminal will support all features.) An explanation of negative cursor function values is given below:

@(-1)	Generates the clear-screen character; clears the screen and positions the cursor at 'home' (upper left corner of the screen).
@(-2)	Positions the cursor at 'home' (upper left corner).
@(-3)	Clears from cursor position to the end of the screen.
@(-4)	Clears from cursor position to the end of the line.
@(-5)	Starts blinking on subsequently printed data.
@(-6)	Stops blinking.
@(-7)	Initiates 'protect' field. All printed data will be 'protected', that is, cannot be written over.
@(-8)	Stops 'protect' field.
@(-9)	Backspaces the cursor one character.
@(-10)	Moves the cursor up one line.

```
@(-11)  Slave printer on.
@(-12)  Slave printer off.
@(-13)  Start reverse video.
@(-14)  Stop reverse video.
@(-15)  Start underline.
@(-16)  Stop underline.
@(-17)  Enable protect mode.
@(-18)  Disable protect mode.
@(-19)  Cursor forward.
@(-20)  Cursor down.
```

Examples of the use of cursor control:

Correct Use	Explanation

`X = 7`
`Y = 3`
`PRINT @(X,Y): Z`
	Prints the current value of variable Z at column position 7 of line 3.

`Q = @(3): "HI"` `PRINT Q`	Prints "HI" at column position 3 of current line.
`A = 5` `PRINT @(A,A+5):A`	Prints the value 5 at column position 5 of line 10.
`PRINT @(-1)`	Clears the screen and positions the cursor at 'home' position.

Incorrect Use	Explanation
`PRINT @: "HI"`	Expression missing after @.
`PRINT @(,10)`	First expression is missing.
`X = (200,-92):  I`	Expression values are out of range.
`PRINT @(-11)`	Cursor control character not defined.

8.3.2 FORMAT STRINGS: NUMERIC AND FORMAT MASK CODES

Variable values may be formatted via the use of format strings. A format string immediately following a variable name or expression specifies that the value will be formatted as specified by the characters within the format string. The format string may also be used directly in conjunction with the PRINT statement.

The format string uses the same subroutines as the ACCESS Mask Conversion Code. It may be used to format both numeric and non-numeric strings. The format string has the following general form:

```
variable = variable "{j}{n}{m}{Z}{,}{c}{$}{(format mask)}"
```

The entire format string is enclosed in quotes. If the format mask is used, it is enclosed in parentheses within the quotes.

The entire format string may be used as a literal, or it may be assigned to a variable. In either case, the format string or variable immediately follows the variable it is to format.

The numeric mask code (Tables 8-1 and 8-2) is represented by the symbols: j, n, m, Z, comma (,), c and $, which control justification, precision, scaling and credit indication. The format mask code controls field length and fill characters.

The formatted value may be assigned to the same variable or to a new variable (as shown in the general form), or it may be used in a PRINT statement form:

```
PRINT X"format string"
```

The format mask code may be used separately or in conjunction with the numeric mask.

The format mask code (Tables 8-1 and 8-2) is enclosed in parentheses and may consist of any combination of format characters and literal data.

The field length specified ('n') should not exceed 99. The format characters are "#", "*", or "%", optionally followed by a numeric such as "#3" or "%5".

Any other character in the format field, including parentheses, may be used as a literal character.

Note that if a dollar sign is placed outside of the format mask, it will be output immediately before the value, regardless of the filled mask. If a dollar sign is used within the format field, it will be output in the leftmost position regardless of the filled field.

Table 8-1. Explanation of the Format String Codes

Mask Code	Explanation
Numeric	
j	specifies justification. "R" specifies right justification. "L" specifies left justification (default). If "D" is specified in this field, a standard system date conversion will be performed.
n	is a single numeric digit defining the number of digits to print out following the decimal point. If n=0, the decimal point will not be output following the value.
m	is an optional 'scaling factor' specified by a single numeric digit which 'descales' the converted number by the 'mth' power of 10. Because BASIC assumes 4 decimal places (unless otherwise specified by a Precision Statement), to descale a number by 10, m should be set to 5, to descale a number by 100, m should be set to 6, etc.
Z	specifies the suppression of leading zeros. Optional.
,	inserts commas between every thousands position of the value. Optional.
c	The following five symbol parameters are Credit Indicators. Optional. C causes the letters 'CR' to follow negative values and causes two blanks to follow positive or zero values. D causes the letters 'DB' to follow positive values; two blanks to follow negative or zero values. M causes a minus sign to follow negative values; a blank to follow positive or zero values. E causes negative values to be enclosed with a "<.....>" sequence; a blank follows positive or zero values. N causes the minus sign of negative values to be suppressed.
$	appends a dollar sign to the leftmost position of the value, prior to conversion. Optional.
Format	
#n	specifies that the data is to be filled on a field of 'n' blanks.
*n	specifies that the data is to be filled on a field of 'n' asterisks.
%n	specifies that the data is to be filled on a field of 'n' zeros. (It will force leading zeros into a fixed field.)

Any other character, including parentheses, may be used as a field fill.

Table 8-2. General Form and Summary of Format String Codes

General Form:

variable = variable"{j}{n}{m}{Z}{,}{c}{$}{(format mask)}"

Mask Code	Implemented as	Meaning
Numeric		
j	R or L D	Right or left justification (default is left justification). D for date conversion.
n	Single numeric	Number of decimal places.
m	Single numeric	'Descaling' factor.
Z	Z	Suppress leading zeros.
,	,	Insert commas every thousands position.
c	C,D,M, or E	Credit indicators.
$	$	Outputs dollar sign before value.
Format (enclosed in parentheses)		
$	$	Outputs a dollar sign in the leftmost position of field.
#n	#10	Fills data on a field of 10 blanks.
%n	%10	Fills data on a field of 10 zeros.
*n	*10	Fills data on a field of 10 asterisks, or on a field of any other specified character.

NOTE: If a dollar sign is placed outside of the format mask, it will be output just prior to the value, regardless of the filled field. If a dollar sign is used within the format mask, it will be output in the leftmost position regardless of the filled field.

Examples of the use of format strings:

Unconverted String (X)	Format String	Result (V)
X = 1000	V = X"R26"	10.00
X = 1234567	V = X"R27,"	1,234.57
X = -1234567	V = X"R27,E$"	$<1234.57>
X = 38.16	V = X"1"	38.2
X = -1234	V = X"R25$,M(*10#)"	***$123.40-
X = -1234	V = X"R25,M($*10#)"	$****123.40-
X = -1234	V = X"R25,M($*10)"	$***123.40-
X = 072458699	V = X"L(###-##-####)"	072-45-5866
X = 072458699	V = X"L(#3-#2-#4)"	072-45-5866
X = SMITH, JOHANNSEN	V = X"L((#13))"	(SMITH, JOHANN)

Incorrect Use	Explanation
V = X"MR26"	MR and ML is the code for ACCESS Mask conversions. In BASIC, use only the R or L.
V = X"RL26"	Both right and left justification cannot be used.
V = X"R212"	The descaling factor may only be a single numeric digit. If necessary, the Precision may be set to zero. This will eliminate decimal places so that a descaling factor of 2 will descale by 100, etc.
V = X"L(#100)"	Fill field should not exceed 99 characters.
V = X"R(9%)"	Format code characters must precede the numeric.
V = X"L*32"	Format string must be enclosed in parentheses.

8.3.3 HEADING AND FOOTING STATEMENTS

The HEADING statement causes the specified text string to be printed as the next page heading. The FOOTING statement causes the specified text string to be printed at the bottom of each page. The HEADING statement has the following general form:

```
HEADING "text{'options'}{text{'options'}}"
```

The first HEADING statement executed will initialize the page parameters. Subsequently, the heading literal data may be changed by a new HEADING Statement, and the new heading will be output at the beginning of the next page. The special heading option characters listed below may be used as a part of a HEADING string expression.

Options	Explanation
C	Centers the line
D	Prints current date
L	Carriage return and line feed
N	No stop at end of page
P{n}	Prints current page number, right justified, in field of 4 (or n) blanks.
PN{n}	Same as above, left justified
T	Prints current time and date

When used, these special characters will be converted and printed as part of the heading. Option characters are enclosed in single quotes. For example:

```
HEADING "INVENTORY LIST 'T' PAGE 'PL'"
```

will print as:

```
INVENTORY LIST 13:30:15  02 NOV 1983  PAGE 1
```

The words "INVENTORY LIST", followed by the current time and date, followed by the word "PAGE", followed by the current page number, followed by a carriage return and line feed will appear at the top of each page. Page numbers are assigned in ascending order starting with page 1.

The FOOTING statement has the following general form:

```
FOOTING "text{'options'}{text{'options'}}"
```

The same set of special option characters may be used in the FOOTING as in the HEADING.

The footing literal data may be changed at any time in the BASIC program by another FOOTING statement; this change will take effect when the end of the current page is reached.

Examples of the use of HEADING and FOOTING statements:

Correct Use	Explanation
HEADING "TIME & DATE: ´TL´"	The text "TIME & DATE:" will be printed followed by the current time and date plus a carriage return and line feed.
HEADING "PAGE ´PL´"	The text "PAGE" will be printed followed by the current page number and a carriage return and line feed.
FOOTING "´LTPL´"	The following footing will be printed: the current time, date and page.

Incorrect Use	Explanation
HEADING "OUTPUT	Terminating quote mark is missing.
HEADING "ONE", "TWO"	Only one expression is allowed in the heading statement.
HEADING	must be enclosed in quotes.

8.3.4 PAGE STATEMENT

The PAGE statement causes the current output device to page, and causes the heading specified by the most recent HEADING statement to be printed as a page heading. The page number may optionally be reset by the PAGE statement. The general form of PAGE statement:

```
PAGE {expression}
```

The number of print lines per page is controlled by the current TERM command. If a FOOTING statement has also been used, the PAGE statement will cause the footing to be printed out at the bottom of the page. If only a footing is desired, a null heading should be assigned. Headings and/or footings must be assigned before the PAGE statement is encountered.

If the PAGE statement has the optional expression, the expression is evaluated and the resulting number becomes the next page number used. If a footing is in effect at the time that the page number is changed, the footing will be printed with a page number one less than the evaluated expression.

Examples of the use of PAGE statement:

Correct Use	Explanation
HEADING "ANNUAL STATISTICS" FOOTING "XYZ CORPORATION" PAGE	The PAGE statement will cause both the specified heading and footing to be printed out when the paging is executed.
PAGE 1	This statement will cause the current footing if any, to print (with a page number of 0), and the current heading, if any to print with a page number of 1.
PAGE X+Y	The current footing and heading will be output, and the page number set to the evaluated result of X+Y.

Incorrect Use	Explanation
FOOTING "ANNUAL STATISTICS" PAGE	The footing will be printed out only if a HEADING statement has been used. (The HEADING statement may be null.)

8.3.5 CURRENT TIME AND DATE: TIME, DATE, AND TIMEDATE FUNCTIONS

The TIME function returns the internal time of day. The DATE function returns the current internal date. The TIMEDATE function returns the current time and date in external format. The general form of these functions:

```
TIME()
DATE()
TIMEDATE()
```

The TIME function returns the string value containing the internal time of day. The internal time is the number of seconds past midnight. For example:

```
X = TIME()
```

This statement assigns the string value of the internal time to variable X.

The DATE function returns the string value containing the internal date. The internal date is the number of days since December 31, 1967. For example:

```
A = DATE()
```

This statement assigns the string value of the internal date to variable A.

The TIMEDATE function returns the string value containing the current time and date in the external format. This format is:

```
HH:MM:SS  DD  MMM   YYYY
```

where: HH = hours
MM = minutes
SS = seconds
DD = day
MMM = month
YYYY = year

For example, the following statement assigns the string value of the current time and date to variable B:

```
B = TIMEDATE()
```

Examples of the use of TIME, DATE, and TIMEDATE:

Correct Use	Explanation
A = TIME()	Assigns string value of current internal time to variable A.
IF TIME() > 1000 THEN GOTO 10	Branches to statement 10 if more than 1000 seconds have passed since midnight.
Q = DATE()	Assigns string value of current internal date to variable Q.
PRINT TIMEDATE()	Prints the current time and date in the external format.
WRITET DATE() ELSE STOP	Writes the string value of the current internal date onto a magnetic tape record.

Incorrect Use	Explanation
B = TIME(5)	Expression is not allowed in TIME function.
DATE()	DATE() is a function and may therefore not appear as a stand alone statement.
X = TIMEDATE	"()" is missing.
X = TIME DATE()	A space must not appear between "TIME" and "DATE".

8.3.6 QUERYING CURRENT VALUE OF SYSTEM FUNCTIONS: SYSTEM

The SYSTEM function gives the PICK programmer the ability to determine the current value or status of various system functions. The general format of the SYSTEM function is:

```
SYSTEM(expression)
```

where expression is a number from 1 through 11 which specifies one of the system function queries shown below.

Expression	System Function Queried
1	PRINTER ON statement. Returns a 1 if in effect, a 0 if not.
2	TERM statement. Returns page width defined.
3	TERM statement. Returns page length defined.
4	TERM statement. Checks number of lines on page defined. Returns number of lines remaining to be printed on current page.
5	Page counter. Returns current page number.
6	Line counter. Returns number of last line printed.
7	TERM statement. Returns terminal type code defined.
8	Returns block size at which tape was last attached. Returns 0 if not attached.
9	Returns current CPU millisecond count.
10	PROC stack. Checks current STON. Returns a 1 if stack on; returns a 0 if stack off.
11	LIST function. Returns a 1 if a LIST is active; returns 0 if not.
12	Returns time in milliseconds.
13	Forces RQM and returns a 1.
14	Returns number of bytes in terminal input buffer.
15	Returns current verb options as a character string.
16	Returns current level of nested EXECUTE statements.
17	Returns error message numbers for TCL commands executed by EXECUTE statement with message numbers separated by an attribute mark.

Note that system function queries 4, 5, and 6 may only be used if a BASIC HEADING/FOOTING command is currently in effect.

8.3.7 INPUT AND OUTPUT CONVERSION: ICONV AND OCONV

The ICONV and OCONV functions provide the PICK input and output conversion capabilities to the BASIC programmer. The general form of these functions:

```
ICONV("string","input-conversion")

OCONV("string","output-converson")
```

The ICONV "input-conversion" specifies the type of input conversion to be applied to the string value resulting from the first "string". The resulting value is always a string.

The OCONV "output-conversion" specifies the type of output conversion to be applied to the string value resulting from the first "string". The resulting value is always a string.

The input and output conversion operations specified may include any one of the following:

D	Convert date to internal format (for ICONV function) or to external format (for OCONV function).
MT	Convert time.
MCDX	Convert ASCII to hexadecimal (for ICONV function) or convert hexadecimal to ASCII (for OCONV function).
T	Convert by table translation.

For a detailed treatment of these and other conversion capabilities, refer to the ACCESS Manual.

WARNING: The ACCESS 'F' conversion cannot be called by these functions. The ACCESS 'MR' and 'ML' conversion may be called by using the Format string 'j' parameter (see Section 8.3.2, Format Strings). This performs the same function and is preferable to using the ICONV or OCONV functions in this case.

Examples of the use of input, output conversions:

Correct Use	Explanation
IDATE = ICONV("2-11-83","D")	Assigns the string value "5785" (the internal date) to the variable IDATE.
A = "5785" B = "D" XDATE = OCONV(A,B)	Assigns the string value "02 NOV 1983" (the external date) to the variable XDATE.

Incorrect Use	Explanation
DATA = ICONV("ABC", "D")	"ABC" is not a legal expression (i.e., it is not a date in external format).
X = OCONV ("123")	Second expression is missing.
B = OCONV("123","X")	"X" is not a legal conversion specification.

8.3.8 FORMAT CONVERSION: ASCII, EBCDIC, CHAR, AND SEQ FUNCTIONS

The ASCII function converts a string value of EBCDIC to ASCII. The EBCDIC function converts a string value from ASCII to EBCDIC. The CHAR function converts a numeric value to its corresponding ASCII character. The SEQ function converts an ASCII character to its corresponding numeric value. The general form of these functions:

```
ASCII(expression)
EBCDIC(expression)
CHAR(expression)
SEQ(expression)
```

The string value of the ASCII expression is converted from EBCDIC to ASCII. For example:

```
A = ASCII(B)
```

The EBCDIC function performs the inverse of ASCII. The string value of the expression is converted from ASCII to EBCDIC. For example:

```
B = EBCDIC(A)
```

The CHAR function converts the numeric value specified by the expression to its corresponding ASCII character string value. For example, the following statement assigns the string value for an Attribute Mark to the variable AM:

```
AM = CHAR(254)
```

The SEQ function performs the inverse of CHAR. The first character of the string value is converted to its corresponding numeric value. The following example will print the number 49:

```
PRINT SEQ("1")
```

For a list of ASCII codes, refer to Appendix A.

Examples of the use of the format conversion functions:

Correct Use	Explanation
READT X ELSE STOP Y = ASCII(X)	Reads a record from the magnetic tape unit and assigns value to variable X. Assigns ASCII value of record to variable Y.
B = EBCDIC(A)	Assigns the EBCDIC value of variable A to variable B.
SM = CHAR(255)	Assigns the string value for a Segment Mark to variable SM.
X = 252 SVM = CHAR(X)	Assigns the string value for a Secondary Value Mark to variable SVM.
DIM C(50) S = 'THE GOOSE FLIES SOUTH' FOR I=1 TO LEN(STRING) C(I) = SEQ(S[I,1]) NEXT I	Encodes in vector C elements the decimal equivalents of individual characters of character string S.

Incorrect Use	Explanation
ASCII(VAL) = S	Intrinsic functions may not appear on the left side of the equality sign.
Q5 = EBCDIC "A"	Parentheses are missing around "A".
Z = CHAR("C")	The expression in the CHAR function must be numeric.
EL = SEQ(P)	A character expression in the SEQ function must be a string enclosed in quotes.

9 debugging BASIC programs

9.1 THE BASIC DEBUGGER

The BASIC Symbolic Debugger facilitates the debugging of new BASIC programs and the maintenance of existing BASIC programs. When a BASIC program is compiled, a symbol table item is automatically generated unless the suppress option (S) has been used. This item is used by the BASIC Debugger to reference symbolic variables during program execution. Therefore, the symbol table must be on file in order to use the BASIC Debugger effectively.

The BASIC Debugger may be entered at execution time by 1) depressing the BREAK key, or 2) using the 'D' (debug) option with the RUN verb. Once the BASIC Debugger has been entered, it will indicate the source code line number about to be executed and will prompt for commands with an asterisk (*) as opposed to the System Debugger prompt: '!' and the TCL prompt: '>'. The user has at his disposal, the following general capabilities:

1. Controlled stepping through execution of program by way of single or multiple steps.

2. Transferring control to a specified step (line number).

3. Breaking (temporary halting) of execution on specified line number(s) or on the satisfaction of specified logical conditions.

4. Displaying and/or changing any variable(s), including dimensioned variables.

5. Tracing variables.

6. Conditional entry to the System Debugger.

7. Directing output (terminal/printer).

8. Stack manipulation (displaying and/or popping the stack).

9. Displaying of specified (or all) source code line(s).

It should be noted that a user now requires SYS2 privileges to use the BASIC debugger. This prevents users from making unauthorized changes to data during reporting and data entry Debugger commands and functions are summarized below:

Basic Debugger Function	Related Command
1. Set breakpoint on logical condition where 'o' is <,>,=,#; 'v' is variable; 'c' is condition to be met; or 'n' is line number where preceded by B$o.	Bvoc{&voc} or B$on
2. Display breakpoint table	D
3. Escape to System Debugger	DEBUG or DE
4. Single/multiple step execution	En
5. End program execution and return to TCL	END
6. Proceed from breakpoint to (specified line 'n')	G, Gn
7. Remove all breakpoints (specified breakpoint 'n')	K, Kn
8. Display source code current line, 'n' lines from current one, number of lines from m-n, all lines	L, Ln, Lm-n, L*
9. Switch output from terminal to printer	LP
10. Pass one breakpoint before stopping, or pass 'n'+1 breakpoints	N, Nn
11. Logoff	OFF
12. Inhibit output. Printer-close output to spooler	P, PC
13. Pop return stack. Display return stack.	R, S
14. Switch turns trace table on/off. Trace specified variable 'v'.	T, Tv
15. Remove all traces. Remove specified trace.	U, Un
16. Request symbol table	Z
17. Display next line number of source code	$
18. Print value of variable 'v', of element 'x' in array 'm', of element 'x,y' in matrix 'm', of entire array 'm', entire symbol table.	/v, /m(x), /m(x,y) /m, /*
19. Set window, remove window setting	[x,y], [
20. Replace symbolic variable	%x,y
21. Display program name, line number and message whether object code "verifies".	?

9.2 USING THE BASIC DEBUGGER

The following is a step-by-step procedure for using the BASIC DEBUGGER. This will demonstrate only a few of the commands to provide the user with an introductory "feeling" for the use of the BASIC DEBUGGER.

A program "SAMPLE" is shown below followed by steps a user might take to debug it.

```
SAMPLE

001 DIM ARRAY(10) ; * ARRAY HAS 10 SLOTS
002 FOR I = 1 TO 20 ; * BUG: LOOP SPECIFIES 20 PASSES, ARRAY HAS ONLY 10
003   ARRAY(I) = I ; * EACH SLOT WILL BE FILLED WITH A CONSECUTIVE #
004 NEXT I
005 PRINT ARRAY(I)
006 END
```

"SAMPLE" compiles without any errors detected. Once it is run however, it aborts with the error message "ARRAY SUBSCRIPT OUT OF RANGE" and traps to the BASIC DEBUGGER. Supposing that the user cannot find the error, the following steps could be taken for detecting the error using the BASIC DEBUGGER.

1. The user enters the command "Z" to the DEBUGGER prompt character "*". The DEBUGGER responds with "FILE/PROG NAME?", the user enters the file name followed by a space followed by the program name. This allows the DEBUGGER access to the symbol table created during compilation. Alternatively, if the user uses the debug option (D)" during run time, access to the symbol table is already established, and use of the "Z" command is unnecessary.

2. To find out how far in the loop the program progressed, the user looks at the variable "I" by entering "/I". The DEBUGGER responds with "11 =", at which the user may change the value of "I" if desired. The user may then want to look at all of the values in the array by entering "/ARRAY". The DEBUGGER responds with "ARRAY(1)=1=", the user depresses return and the DEBUGGER continues with the next "array slot" (i.e., "ARRAY(2)=2=", etc.). Once "ARRAY(10)=10=" has been reached, the user presses return and the DEBUGGER returns with the "*" prompt. The user knows that the array has only 10 slots and the loop calls for 20, thus, he finds the error. The user may then end the "session" with BASIC DEBUGGER by entering "END" and repair the bug.

A summary of this interaction is:

```
[B17] LINE 3 ARRAY SUBSCRIPT OUT OF RANGE
*13
*[Z] FILE/PROG NAME?[BP/SAMPLE]
*[/I]  [CR]  11=
*[/ARRAY]  [CR]  ARRAY(1)=1=  [CR]
  ARRAY(2)=2=  [CR]
  ARRAY(3)=3=  [CR]
  ARRAY(4)=4=  [CR]
  ARRAY(5)=5=  [CR]
        .
        .
        .
 ARRAY(10)=10=  [CR]
*[CR]
*[END]
```

For purposes of clarity, whatever is entered by the user is shown enclosed in square brackets "[]". These are not part of the commands; they are to distinguish user entry from DEBUGGER response.

A carriage return will return control to the BASIC DEBUGGER prompted by "*" whereas a line-feed will return control to program execution until a breakpoint, an error, or the end of the program is met.

9.2.1 THE TRACE TABLE

The trace table is used for the automatic printout of a specified variable or variables after a break has occurred.

Up to six trace values may be entered in the table. Either the symbolic name, or a line number and variable number may be used to reference the variable. In addition, all variables in the last statement executed may be printed out. The trace table may be alternately turned on and off by use of the "T" return command.

Examples of the use of the trace table are shown below:

Tname	The value of the variable name will be printed out at each breakpoint.
T%10,3	The value of the third variable in line number 10 will be printed out at each breakpoint. If line number 10 contains the statement "A=B+C+D" the value of "C" will be printed.

To delete a variable from the trace table use the "U" command followed by the trace variable to be deleted. For example, to delete the variable name from the table, type in "Uname". "U" by itself followed by a return deletes the entire trace table.

If a program calls an external subroutine, and the BASIC/DEBUGGER has been entered previously, a complete symbol table will be set up for the external subroutine. The table will have 4 breakpoints and 6 variable traces available, as well as pointers to program source and object, which may be set up by the Z command. Breakpoints set up for a subroutine are independent from breakpoints set up in the main program or other subroutines; however, the execution counters (E and N,) are global.

The use of multiple symbol tables allows the programmer to set up different break points and/or variable traces for different subroutines.

9.2.2 BREAKPOINT CONTROL

The "B", "D", and "K" commands are used to set (B)reakpoints, (D)isplay, and (K)ill breakpoints.

The breakpoint table may contain up to four conditions that, when satisfied, will cause a break in execution. Logical expressions and special symbols are used to set the break conditions. They are:

<	less than
>	greater than
=	equal to
#	not equal to
&	used as a logical connector between conditions
$	a special symbol used to indicate that a line number is specified rather than a variable name.

The basic forms of the "B" command are shown below:

```
B variable-name operator expression {& another condition}
B $ operator line-number {& another condition}
```

where ´variable-name´ is a simple variable or an explicitly stated array element and ´expression´ is a variable, constant, or array element. If the variable does not exist or if the wrong Symbol Table is assigned, the message "SYM NOT FND" will be printed. String constants must be enclosed in quotes using the same rules that apply to BASIC literals. Consider the examples:

BTAX=500	Indicates that an execution break should occur when the value of TAX is equal to 500.
B$>15&X=3	Causes program to break when the line number is greater than 15 and if X is equal to 3.

A plus sign will be printed next to the command if it is accepted. When the condition is met, an execution break will occur and the Debugger will halt execution of the program and print *Bn l where ´n´ is one of the four Breakpoint Table entries and ´l´ is the program line number that caused the break.

The general form of the "D" command is:

D

The "D" command will display the trace and breakpoint tables.

The general form of the K command is:

Kn or K

The "K" command is used to delete breakpoint conditions from the table.

A minus sign will be printed next to the command to indicate that an entry has been removed. "Kn" deletes the nth breakpoint condition where 'n' is 1 through 4. K deletes all breakpoint conditions.

Examples of the use of B, D, K commands:

Correct Use	Explanation
BX<42	Sets a break condition to halt execution when X is less than 42.
BADDRESS=''	Breaks when ADDRESS is null.
BDATE=INV.DATE&$=22	Breaks when variable DATE is equal to variable INV.DATE and if the line number is 22.
K2	Kills the second breakpoint condition.
BPRICE(3)=24.98	Sets a break condition to halt execution when the third element of the array PRICE is equal to 24.98. Only individual array elements may be specified.
D	Displays the trace tables (1 through 6) and breakpoint tables (1 through 4).
K	Kills all breakpoint conditions.

9.2.3 EXECUTION CONTROL

The "E", "G", and "N" commands in conjunction with the breakpoint table control the execution of the program under debug control.

The "E" command will allow the execution of a specified number of lines before returning control to the user. The number of statements to be executed is selected by putting a numeric value after "E". For example, "E3" will execute three line statements before returning control to the user. "E" return will turn off the "E" command.

Command	Result
E10	Ten line statements will be executed before control returns to the user.
E	Execution continues until interruption by the user, by a breakpoint, or until program ends.

The "N" command will allow the user to bypass any number of breakpoints before control is passed back to the user, however, the trace table variables will be printed at each breakpoint. "N0" equals 'pass one breakpoint', "N1" equals 'pass two breakpoints', etc., and "N" return will set "N" to "N0".

Command	Result
N3	Four breakpoints are passed, although the trace table values, if present, are printed out at each breakpoint. Control is returned to the user after the fourth breakpoint.
N	One breakpoint is passed.

The "G" command followed by a line number will allow control to be passed to the line number indicated. The "G" return command will cause program to execute the next command from the current line number and it will return control depending on the breakpoint setup. G may be entered in either upper or lower case.

Command	Result
G153	Control passes to line number 153, and thereafter to user.
G	Control passes to next program line, and thereafter to user.

9.2.4 DISPLAYING AND CHANGING VARIABLES

Variables and arrays can be displayed and changed in either decimal or string formats. To display a variable, use the command '/v' where 'v' is a variable. For example, to display the value of the variable 'NAME', you would enter /NAME after the * prompt. The DEBUGGER will respond with the string in the NAME field and an equal sign. If the variable is not to be changed, press return. If the variable is to be changed, put in the new value of the variable desired and press return. To display a complete array, just place the name of the array after the slash. To display one value in the array, use the form '/M(x)' or '/M(x,y)' where 'x' and 'y' are points in the array. The array point may then be changed in the same way as a single variable.

A window may be placed after any variable selection by following the variable with a ';' and the length of the window. For example, to limit the variable name to eight characters, the command '/name;8' would be used. Numeric variables will ignore any window commands.

The symbolic name of the variable may be replaced using the form '%x,y' where 'x' is the line number and 'y' is the number of the variable in that line. Examples of displaying and changing variables:

/CITY IRVINE=	The variable 'city' is displayed but not changed.
/STATE N.Y.=C.A	The variable 'state' is displayed as 'N.Y.' and changed to 'C.A.'
/FIELD(5) 10=	The fifth point in array FIELD is displayed as 10 and not changed.
/*	All the symbols in the symbol table are displayed.

9.2.5 SPECIAL COMMANDS

1. I/O Control. Three commands "PC", "P" and "LP" control I/O.

   The "P" command inhibits all BASIC program output so that the user may look only at the DEBUGGER output. "P" return alternately turns "P" on and off.

   The "LP" command forces all output to the line printer which can be used for a fast trace or hard copy of a trace. "LP" return alternately turns the line printer command on and off.

   The "PC" command is the same as the BASIC printer close command. All data that is waiting to be sent to the printer is output at this time.

2. Source Code Display. The "$" command will print the next line number to be executed.

   The "L" command will display source code lines. "L" will display the current line of source. "Ln" will display line 'n'. "Lm-n" will display lines 'm-n'. "L*" will display the entire source program.

3. Symbol Table. The "Z" command will allow the operator to specify a symbol table. After the user enters the file-name and program name, that symbol table, if present, will be enabled.

4. String Windows. The string window "[n,m]" will cause the output of all variables to be limited to the substring selected. An example:

   X=1234567890

   [3,2] sets the window for the third character position with a string length of two. Any printout of x will be 34. Setting the window length to zero or entering a left square bracket ([) will turn the string window command off.

5. Escape to System Debugger. The "DEBUG" command will pass control to the System Debugger.

6. Termination. The "END" command will terminate the BASIC and DEBUG programs and return control to TCL. "END" may be entered in either upper or lower case.

A "?" will display the program name, the number of the last line executed, and the error message.

9.3 SUMMARY OF THE BASIC DEBUGGER COMMANDS

The following is a summary of all the BASIC Debugger commands and their descriptions.

Command	Description
Bvoc{&voc} or B$on	Sets breakpoint on logical condition where 'o' is logical operator <, >, =, or #; 'v' is variable; 'c' is condition to be met; or 'n' is line number where preceded by B$o.
D	Displays breakpoint and trace tables.
DEBUG	Escape to system debugger.
DE	Short form of DEBUG.
En	Step on N+1 instructions. E [CR] turns mode off.
END or end	End execution of BASIC program and return to TCL.
G or g	Proceed from breakpoint.
Gn or gn	Go to line n.
K	Kills all breakpoint conditions in table set by 'B' command.
Kn	Kills breakpoint condition 'n', where 'n' is the breakpoint number from 1-4.
LP	All output forced to printer reverses status each time LP is selected.
N	Continue through one breakpoint before stopping.
Nn	Continue through n+1 breakpoints before stopping.
OFF or off	Log off.
P	Inhibit BASIC program output.
PC	Printer close - output to spooler.
R	Pops return stack.
S	Display subroutine stack.
T	Turns breakpoint trace table alternately off and on.
Tv	Set variable 'v' in trace breakpoint table.

Command	Description
U	Remove all breakpoint trace table variables set by 'T' command.
Uv	Remove breakpoint trace variable 'v' from table.
Z	Request symbol table.
$	Current statement number.
/V	Print value of a variable 'V'.
/m(x)	Print value of a point 'x' in array 'm'.
/m(x,y)	Print value of point 'x,y' in array 'm'.
/m	Print the entire array where 'm' is the array.
/*	Dump entire symbol table.
[x,y]	String window where 'x' equals the start of the string and 'y' equals the length of the string. This command affects all outputs of variables and has no effect on input.
[	Removes string window (setting string length to zero has the same effect).
=	Equal sign prints out after the printing of a variable in any slash command except '/m'. The value of the variable may be changed at this point.
%x,y	Replaces symbolic variable; 'x' is line number and 'y' is number of variable in that line.
?	Displays program name, last line number executed, and reports whether or not object code verifies with checksum generated at compile time.

NOTES:

1. Carriage return terminates all controls.
2. A line feed equals G [CR]
3. BREAK key breaks to BASIC Debugger from BASIC program at end of line.
4. BASIC Debugger prompts with '*'.

9.4 BASIC DEBUGGER MESSAGES

The following informative, warning, or error messages are used by the BASIC Debugger.

Message	Description
*E x	Single step breakpoint at line number 'x'.
*Bn x	Table breakpoint at line number 'x', 'n' equals number of breakpoint.
*V=x	Value of variable at breakpoint.
*Nvar	Variable not found in statement.
CMND?	Command not recognized.
NSTAT#	Statement number out of range of program.
SYM NOT FND	Symbol not found in table.
UNASSIGNED VAR	Variable not assigned a value.
STACK EMPTY	The subroutine return stack is empty.
STACK ILL	Illegal subroutine return stack format.
TBL FULL	Trace or break table full.
ILLGL SYM	Illegal symbol.
NOT IN TBL	Not in trace break table.
NO SYM TAB	Symbol table not in file.

general coding techniques and sample programs 10

10.1 GENERAL CODING TECHNIQUES

The PICK system uses standard attribute and value delimiters. These should be defined once in the initialization portion of the program, and then referenced by their variable name. For example:

```
    AM = CHAR(254)    Attribute Mark
    VM = CHAR(253)    Value Mark
   SVM = CHAR(252)    Secondary Value Mark
```

Cursor positioning should be controlled by the following PRINT statements using the @ functions:

Erase screen	PRINT @(-1)	Stop blink	PRINT @(-6)
Home	PRINT @(-2)	Start protect	PRINT @(-7)
Clear to end of screen	PRINT @(-3)	Stop protect	PRINT @(-8)
Clear to end of line	PRINT @(-4)	Backspace	PRINT @(-9)
Start blink	PRINT @(-5)	Up 1 line	PRINT @(-10)

The OPEN statement is time consuming and should be executed as few times as possible. All files should be opened to file variables at the beginning of the program; access to the files can then be performed by referencing the file variables. The size of programs can be reduced, with a corresponding increase in overall system performance, by reducing the amount of literal storage.

```
200 PRINT 'RESULT IS ':A+B
210 PRINT 'RESULT IS ':A-B
220 PRINT 'RESULT IS ':A*B
230 PRINT 'RESULT IS ':A/B
```

These statements should have been written as follows:

```
    MSG = 'RESULT IS'
    .
    .
    .
200 PRINT MSG:A+B
210 PRINT MSG:A-B
220 PRINT MSG:A*B
230 PRINT MSG:A/B
```

Operations should be predefined rather than repetitively performed. This operation, for example:

```
X = SPACE(9-LEN(OCONV(COST,'MCA')):OCONV(COST,'MCA')
```

should have been written as:

```
E=OCONV(COST,'MCA')
X=SPACE(9-LEN(E)):E
```

In the same context, the following operation:

```
FOR I=1 TO X*Y+Z(20)
 .
 .
 .
NEXT I
```

should have been written as:

```
TEMP=X*Y+Z(20)
FOR I=1 TO TEMP
 .
 .
 .
NEXT I
```

The following LOOP construct could be used to access an unknown number of multivalues from an attribute (including null values):

```
VM=CHAR(253)
READV ATTR FROM ID, ATTNO ELSE STOP
VND=0
LOOP
   VND=VNO+1
   VALUE=FIELD(ATTR,VM,VNO)
WHILE COL2() #0 DO
   PRINT VALUE
REPEAT
```

10.2 SAMPLE PROGRAMS

Examples to demonstrate the use of different BASIC programs are presented in this section.

PYTHAG

```
*******************************************************
*  THIS PROGRAM FINDS PYTHAGOREAN TRIPLES
*******************************************************
      PRINT
      PRINT 'SOME PYTHAGOREAN TRIPLES ARE:'
      PRINT
      FOR A=1 TO 40
         FOR B=1 TO A-1
            CC=A*A+B*B
            GOSUB 50
            IF C = INT(C) THEN PRINT B,A,C
         NEXT B
      NEXT A
      STOP
*  SQUARE ROOT SUBROUTINE
50    C=CC/2
      FOR I=1 TO 20
         X=(C+CC/C)/2
         IF C = X THEN RETURN
         C=X
      NEXT I
      RETURN
  END
```

GUESS

```
*********************************************************
*  THIS PROGRAM IS A GUESSING GAME
*********************************************************
      HEADING ''
      HISSCORE=0; YOURSCORE=0
10    PAGE
      PRINT 'GUESS NUMBERS BETWEEN 0 AND 100'
      PRINT 'MACHINE:':HISSCORE:'    YOU:':YOURSCORE:
      PRINT
      NUM=RND(101)
      FOR I=1 TO 6
         PRINT 'GUESS ':I:'  ':
         INPUT GUESS
         IF GUESS=NUM THEN
            PRINT
            PRINT 'CONGRATULATIONS, YOU WON!'
            YOURSCORE=YOURSCORE+1
            GOTO 60
         END
         IF GUESS<NUM THEN PRINT 'HIGHER'
         IF GUESS>NUM THEN PRINT 'LOWER'
      NEXT I
      PRINT
      PRINT 'YOU LOST YOU DUMMY, YOUR NUMBER WAS ':NUM
      HISSCORE=HISSCORE+1
60    PRINT
      PRINT 'AGAIN?':
      INPUT X
      IF X = 'NO' THEN STOP
      GOTO 10
   END
```

INV-INQ

```
*********************************************************************
*    THIS PROGRAM QUERIES AN INVENTORY FILE.
*    IT READS THE DICTIONARY OF FILE 'INV' TO GET THE ATTRIBUTE
*    NUMBERS OF 'DESC' (DESCRIPTION) AND 'QOH' (QUANTITY-ON-HAND).
*    THE PROGRAM THEN PROMPTS THE USER FOR A PART-NUMBER WHICH
*    IS THE ITEM-ID OF AN ITEM IN 'INV' AND USES THE ATTRIBUTE
*    NUMBERS TO READ AND DISPLAY THE PART DESCRIPTION AND
*    QUANTITY ON HAND.  THE PROGRAM LOOPS UNTIL A NULL PART
*    NUMBER IS ENTERED.
*********************************************************************
*
*--- GET ATTRIBUTE DEFINITIONS FROM DICTIONARY OF INVENTORY FILE
     OPEN 'DICT','INV' ELSE PRINT 'CANNOT OPEN "DICT INV"'; STOP
     READV DESC.AMC FROM 'DESC',2 ELSE PRINT 'CANT READ "DESC" ATTR'; STOP
     READV QOH.AMC FROM 'QOH',2 ELSE PRINT 'CANT READ "QOH" ATTR'; STOP
*--- OPEN DATA PORTION OF INVENTORY FILE
     OPEN '','INV' ELSE PRINT 'CANNOT OPEN "INV"'; STOP
*--- PROMPT FOR PART NUMBER
100  PRINT
     PRINT 'PART NUMBER ':
     INPUT PN
     IF PN = '' THEN PRINT, '--DONE--'; STOP
     READV DESC FROM PN,DESC.AMC ELSE PRINT 'CANT FIND THAT PART';GOTO 100
     READV QOH FROM PN,QOH.AMC ELSE QOH=0
*--- PRINT DESCRIPTION AND QUANTITY-ON-HAND
     PRINT 'DESCRIPTION - ': DESC
     PRINT 'QTY-ON-HAND - ': QOH
     PRINT
     GOTO 100
  END
```

FORMAT

```
*************************************************************************
*  THIS PROGRAM FORMATS A BASIC PROGRAM TO
*  DISPLAY BLOCK STRUCTURING BY INDENTING LINES.
*************************************************************************
*--- DEFINITIONS
10   SP = 6                    ;* LEFT MARGIN COLUMN NUMBER
     ID = 3                    ;* NUMBER OF SPACES TO INDENT
*--- INITIALIZATION
     SPX = SP
     LINE.NO = 0
*--- INPUT FILE NAME AND PROGRAM NAME
     PRINT
     PRINT
     PRINT 'DATA/BASIC FILE NAME     - ':; INPUT FILE
     IF FILE =, '' THEN STOP
     OPEN '',FILE ELSE PRINT 'CANNOT OPEN FILE - ': FILE; GOTO 10
     PRINT 'BASIC PROGRAM NAME - ':; INPUT NAME
     IF NAME = '' THEN GOTO 10
     NEW ITEM = ''
     READ ITEM FROM NAME ELSE
        PRINT 'CANNOT FIND THAT PROGRAM'
        GOTO 10
     END
*--- GET NEW LINE, IF NONE - THEN DONE
100  LINE.NO = LINE.NO + 1
     LINE = EXTRACT(ITEM,LINE.NO,0,0)
     IF LINE = '' THEN
        WRITE NEWITEM ON NAME
        PRINT; PRINT; PRINT '--DONE--'; GOTO 10
     END
     LABEL = ''
*--- STRIP OFF LEADING/TRAILING SPACES
200  IF LINE[1,1] = ' ' THEN LINE = LINE[2,32767]; GOTO 200
210  IF LINE[LEN(LINE),1] = ' ' THEN LINE = LINE[1,LEN(LINE)-1]; GOTO 210
*--- LOOK FOR A COMMENT ('*', '!', OR 'REM')
     IF LINE[1,1] = '*' THEN GOTO 1500
     IF LINE[1,1] = '!' THEN GOTO 1500
     IF LINE[1,3] = 'REM' THEN GOTO 1500
*--- LOOK FOR 'FOR'
     IF LINE[1,4]='FOR ' AND INDEX(LINE,'NEX ',1)>0 THEN GOTO 2000
     IF LINE[1,4]='FOR ' AND INDEX(LINE,'NEXT  ',1)=0 THEN GOTO 1000
*--- LOOK FOR 'END'
     IF LINE = 'END' THEN GOTO 1100
     IF LINE[1,4] = 'END ' THEN
        IF LINE[LEN(LINE)-4,5] = ' ELSE' THEN GOTO 1200
     END
```

```
*--- LOOK FOR 'NEXT'
     IF LINE[1,5] = 'NEXT ' THEN GOTO 1100
*--  EXTRACT LEADING NUMERIC LABEL
     IF LINE[1,1] MATCHES '1N' THEN
        L = 2
300     IF LINE[L,1] MATCHES '1N' THEN L=L+1; GOTO 300
        LABEL = LINE[1,L-1)]
        LINE = LINE[L,32767]
        GOTO 200
     END
*--- LOOK FOR LINE ENDING IN ' ELSE' OR ' THEN' ('IF' OR 'READ')
     X = LINE[LEN(LINE)-4,5]
     IF X = ' THEN' THEN GOTO 1000
     IF X = ' ELSE' THEN GOTO 1000
*--- THIS IS JUST ANOTHER LINE, THEREFORE NO CHANGE
     GOTO 2000
*--- INDENT ON SUBSEQUENT LINES
1000 SP = SP + ID
GOTO 2000
*--- OUTDENT ON THIS AND SUBSEQUENT LINES
1100 SP = SP - ID
*--- OUTDENT THIS LINE ONLY
1200 SPX = SPX - ID
     GOTO 2000
*--- PRINT WITH NO INDENTATION
1500 SPX = 0
*--- WRITE NEW LINE
2000 NEW.LINE = LABEL : STR(' ',SPX-LEN(LABEL)) : LINE
     PRINT NEW.LINE
     NEWITEM = REPLACE(NEWITEM,LINE.NO,0,0,NEW.LINE)
     SPX = SP
     GOTO 100
  END
```

LOT-UPDATE

```
    ************************************************************
    *    THIS PROGRAM UPDATES DATA ON LOTS IN A HOUSING TRACT.
    *    ITEM-ID'S IN "LOT" FILE ARE TRACT.NAME*LOT.NUMBER
    ************************************************************
    100*  INITIALIZATION
           PROMPT '='
           CLEAR
           DIM DESC(30),TYPE(30)
           OPEN 'DICT','LOT' ELSE
              PRINT "CAN'T OPEN DICT LOT"
              STOP
           END
*
    200*  GET DESCRIPTIONS, CONVERSIONS
           FOR I = 1 TO 30
              READ DICT.ITEM FROM 1 ELSE
                 PRINT "DICTIONARY ITEM '":I:"' NOT FOUND"
                 GOTO 250
              END
              D = EXTRACT(DIC.ITEM,3,0,0)            ;* S/NAME--DESCRIPTION
              IF D # '' THEN DESC(I) = D:STR('.',15-LEN(D))
              IF C[1,2] = 'MD' THEN
                 TYPE(I) + 'NUM'
                 GOTO 250
              END
              IF C[1,1] = 'D' THEN TYPE(I) = 'DATE'
    250*
           NEXT I
    *
    *
           OPEN '','LOT' ELSE
              PRINT "CAN'T OPEN LOT FILE."
              STOP
           END
    *
    300*  GET THE TRACT NAME
           PRINT
           PRINT "TRACT NAME.....":
           INPUT TRACT
           IF TRACT = 'STOP' OR TRACT = 'END' THEN STOP
           IF TRACT = '' THEN GOTO 300
           READ INFO FROM TRACT ELSE
              PRINT "TRACT '":TRACT:"' LOT ON FILE"
              GOTO 300
           END
```

```
*
400*  GET A VALID LOT NUMBER
       PRINT
       PRINT "LOT NUMBER.....":
       INPUT NUMBER
       IF NUMBER = '' THEN GOTO 400
       IF NUMBER = 'END' OR NUMBER = 'STOP' THEN GOTO 300
       IF NUM(NUMBER) = 0 THEN
          PRINT "MUST BE A NUMBER"
          GOTO 400
      END
      NUMBER = TRACT:'*':NUMBEZR
      READ ITEM FROM NUMBER ELSE
      ITEM = ''
      PRINT "NEW LOT"
   END
*
450*
      NOT.SOLD = 0
      FOR I = 1 TO 30
            GOSUB 1000                          ;* UPDATES THE ('TH ATTRIBUTE
         IF I = 10 THEN
            IF EXTRACT(ITEM,10,0,0) = '' THEN
               NOT.SOLD = 1
               I = 19
            END
         END
*
         IF I = 21 THEN
            IF NOT.SOLD THEN GOTO 500
         END
      NEXT I
*
      VERIFY DATA & STORE
      PRINT
      PRINT"        OK       ":
      INPUT OK
      IF OK = '' THEN
            WRITE ITEM ON NUMBER
            GOTO 400
      END
      IF OK = 'L' THEN
         PRINT
         FOR L = 1 TO 30
            ATT = EXTRACT(ITEM,I,0,0)
            IF ATT = '' THEN GOTO 550
            PRINT DESC(L):
            IF TYPE(L) = 'DATE' AND NUM(DATE) THEN ATT = OCONV(ATT,'D0')
            IF TYPE(L) = 'NUM' AND NUM(ATT) THEN ATT = 0.01 * ATT
            PRINT ATT 'R################'
```

```
550*
      NEXT L
      GOTO 500
   END
   GOTO 400
*
1000* UPDATE'S THE I'TH ATTRIBUTE OF "ITEM"
      IF DESC(I) = '' THEN RETURN              ;* NOT NEEDED OR NOT FOUND
      PRINT DESC(I):
      CURRENT = EXTRACT(ITEM,I,0,0)
*
      IF TYPE(I) = 'NUM' THEN
1100* NEED A NUMBER (AMOUNT)
         PRINT CURRENT*.01   'R################':
         INPUT RESPONSE
         IF RESPONSE = '' THEN RETURN          ;*JUST LOOKING
         IF RESPONSE = '' THEN
           ITEM = REPLACE(ITEM,I,0,0,'')
           RETURN                              ;* DELETE THIS ATT.
         END
         IF NUM(RESPONSE) = 0 THEN
            PRINT "MUST BE A NUMBER"
            GOTO 1100
         END
            ITEM = REPLACE(ITEM,I,0,0,RESPONSE*100)
            RETURN
        END
*
      IF TYPE(I) = 'DATE' THEN
1200* NEED A DATE
         PRINT OCONV(CURRENT,'D0')  'R##################':
         INPUT RESPONSE
         IF RESPONSE = '' THEN RETURN            ;* JUST LOOKING
         IF RESPONSE = 'T' THEN
            DATE = DATE()
            GOTO 1250
         END
         IF RESPONSE = '' THEN
            ITEM = REPLACE(ITEM,I,0,0,'')        ;*DELETE THIS ATT.
            RETURN
         END
         DATE = ICONV(RESPONSE,'D')
         IF DATE = '' THEN
            PRINT "USE DATE FORMAT 'MONTH/DAY/YEAR/'"
            GOTO 1200
         END
1250*
         ITEM = REPLACE(ITEM,I,0,0,DATE)
         RETURN
      END
```

```
1300* NO NECESSARY FORMATS
      PRINT CURRENT 'R################':
      INPUT RESPONSE
      IF RESPONSE = '' THEN RETURN
      IF RESPONSE = '' THEN RESPONSE = ''
      ITEM = REPLACE(ITEM            I,0,0,RESPONSE)
      RETURN
END
```

ASCII codes A

The ASCII codes used by the PICK System are:

DEC	Hex	Character	DEC	Hex	Character
0	0	NULL	36	24	$
1	1	SOH	37	25	%
2	2	STX	38	26	&
3	3	ETX	39	27	'
4	4	EOT	40	28	(
5	5	ENQ	41	29	)
6	6	ACK	42	2A	*
7	7	BEL	43	2B	+
8	8	BS[1]	44	2C	,
9	9	HT[1]	45	2D	-
10	A	LF[1]	46	2E	.
11	B	VT[1]	47	2F	/
12	C	FF[1]	48	30	0
13	D	CR[1]	49	31	1
14	E	SO	50	32	2
15	F	SI	51	33	3
16	10	DLE	52	34	4
17	11	DC1	53	35	5
18	12	DC2	54	36	6
19	13	DC3	55	37	7
20	14	DC4	56	38	8
21	15	NAK	57	39	9
22	16	SYN	58	3A	:
23	17	ETB	59	3B	;
24	18	CAN	60	3C	<
25	19	EM	61	3D	=
26	1A	SUB	62	3E	>
27	1B	ESC	63	3F	?
28	1C	FS	64	40	@
29	1D	GS	65	41	A
30	1E	RS[1]	66	42	B
31	1F	US[1]	67	43	C
32	20	SPACE	68	44	D
33	21	!	69	45	E
34	22	"	70	46	F
35	23	#	71	47	G

DEC	Hex	Character	DEC	Hex	Character
72	48	H	104	68	h
73	49	I	105	69	i
74	4A	J	106	6A	j
75	4B	K	107	6B	k
76	4C	L	108	6C	l
77	4D	M	109	6D	m
78	4E	N	110	6E	n
79	4F	O	111	6F	o
80	50	P	112	70	p
81	51	Q	113	71	q
82	52	R	114	72	r
83	53	S	115	73	s
84	54	T	116	74	t
85	55	U	117	75	u
86	56	V	118	76	v
87	57	W	119	77	w
88	58	X	120	78	x
89	59	Y	121	79	y
90	5A	Z	122	7A	z
91	5B	[	123	7B	{
92	5C	\	124	7C	:
93	5D	]	125	7D	}
94	5E	^	126	7E	~
95	5F	_	127	7F	DEL
96	60	`	•		
97	61	a	•		
98	62	b	•		
99	63	c	251	FB	SB[2]
100	64	d	252	FC	SVM[2]
101	65	e	253	FD	VM[2]
102	66	f	254	FE	AM[2]
103	67	g	255	FF	SM[2]

[1]For special use on LSI-11 and -12 terminals:

BS	Cursor Backspace	FF	Cursor Forward
HT	Cursor Tab	CR	Cursor Carriage Return
LF	Cursor Down	RS	Cursor Home
VT	Cursor UP	US	Cursor New Line

[2]For special use by PICK:

SB	Start buffer
SVM	Secondary value mark (displays \)
VM	Value mark (displays])
AM	Attribute mark (displays ^)
SM	Segment mark (displays _)

Decimal	Hex	Character	Special Use in PICK
84	54	T	
85	55	U	
86	56	V	
87	57	W	
88	58	X	
89	59	Y	
90	5A	Z	
91	5B	[	
92	5C	\	
93	5D	]	
94	5E	^	
95	5F	_	
96	60	`	
97	61	a	
98	62	b	
99	63	c	
100	64	d	
101	65	e	
102	66	f	
103	67	g	
104	68	h	
105	69	i	
106	6A	j	
107	6B	k	
108	6C	l	
109	6D	m	
110	6E	n	
111	6F	o	
112	70	p	
113	71	q	
114	72	r	
115	73	s	
116	74	t	
117	75	u	
118	76	v	
119	77	w	
120	78	x	
121	79	y	
122	7A	z	
123	7B	{	
124	7C	:	
125	7D	}	
126	7E		
127	7F	DEL	
.			
251	FB	SB	Start buffer
252	FC	SVM	Secondary value mark (displays \)
253	FD	VM	Value mark (displays])
254	FE	AM	Attribute mark (displays ^)
255	FF	SM	Segment mark (displays _)

compiler error messages B

This section presents a list of the error messages which may occur as a result of compiling a BASIC program.

Error No.	Error Message	Cause
B100	COMPILATION ABORTED; NO OBJECT CODE PRODUCED	Compilation errors present.
B101	MISSING "END", "NEXT", "WHILE", "UNTIL", "REPEAT", OR "ELSE"; COMPILATION ABORTED, NO OBJECT CODE PRODUCED	Compilation error present.
B102	BAD STATEMENT	Unrecognizable statement.
B103	LABEL "C" IS MISSING	Label indicated by GOTO or GOSUB was not found.
B104	LABEL "C" IS DOUBLY DEFINED	More than one statement was found beginning with the same label.
B105	"C" HAS NOT BEEN DIMENSIONED	Subscripted variable was not dimensioned.
B106	"C" HAS BEEN DIMENSIONED AND USED WITHOUT SUBSCRIPTS	Dimensioned variable used without subscripts.
B107	"ELSE" CLAUSE MISSING	ELSE clause is missing.
B108	"NEXT" STATEMENT MISSING	NEXT statement is missing in FOR-NEXT loop.
B109	VARIABLE MISSING IN "NEXT" STATEMENT	Iteration variable is missing in NEXT statement.
B110	"END" STATEMENT MISSING	END statement is missing in multiline IF statement.

Error No.	Error Message	Cause
B111	"UNTIL" OR "WHILE" MISSING IN "LOOP" STATEMENT	UNTIL or WHILE clause is missing in a LOOP statement.
B112	"REPEAT" MISSING IN "LOOP" STATEMENT	REPEAT is missing in a LOOP statement.
B113	LINE 'A' TERMINATOR MISSING	Garbage was found following a legal statement, or quote was missing.
B114	MAXIMUM NUMBER OF VARIABLES EXCEEDED	Using the default descriptor size of 10, the maximum number of variables (including array elements) is 3274.
B115	LABEL 'C' IS USED BEFORE THE EQUATE STATEMENT	The equate-variable is referenced before it has been defined.
B116	LABEL 'C' IS USED BEFORE THE COMMON STATEMENT	A common variable was referenced before it was put in common.
B117	LABEL 'C' IS MISSING A SUBSCRIPT LIST	An array is referenced without a subscript list.
B118	LABEL 'C' IS THE OBJECT OF AN EQUATE STATEMENT AND IS MISSING	Label 'C' referenced, but no label 'C' found.
B119	WARNING - PRECISION VALUE OUT OF RANGE - IGNORED!	
B120	WARNING - MULTIPLE PRECISION STATEMENTS - IGNORED!	
B121	LABEL 'C' IS A CONSTANT AND CAN NOT BE WRITTEN INTO	
B122	LABEL 'C' IS IMPROPER TYPE	
B123	THE PROGRAM CONTAINS AN EQUATE WHICH CANNOT BE RATIONALIZED	
B124	LABEL 'C' HAS LITERAL SUBSCRIPTS OUT OF RANGE	

Error No.	Error Message	Cause
B125	LABEL 'C' HAS A JUMP GREATER THAN 32K BYTES	
B126	OBJECT CODE EXCEEDS 65K	
B127	OBJECT CODE AND SYMBOL TABLE EXCEED 65K	
B128	LABEL 'C' EQUATED ARRAY SUBSCRIPT OUT OF RANGE	
B154	FOR STATEMENT WITH NO NEXT STATEMENT	
B199	FORMAT ERROR IN SOURCE FILE DEFINITION	
B850	NODE NOT AVAILABLE	
B853	UNEXPECTED MESSAGE RECEIVED	
B854	NO LAN PROCESSOR ON THIS MACHINE	

run-time error messages C

This section presents a list of the error messages which may occur as a result of executing a BASIC program. Warning messages indicate that illegal conditions have been smoothed over (by making an appropriate assumption) and do not result in program termination. Fatal error messages result in program termination.

Error No.	Error Message	Cause
B1	RUN-TIME ABORT AT LINE A	[FATAL]
B10	VARIABLE HAS NOT BEEN ASSIGNED A VALUE; ZERO USED!	An unassigned variable was referenced. (A value of 0 is assumed.) [WARNING]
B11	TAPE RECORD TRUNCATED TO TAPE RECORD LENGTH!	An attempt was made to write more onto a tape record than the tape record length. (The record is truncated to tape record length.) [WARNING]
B12	FILE HAS NOT BEEN OPENED	File indicated in I/O statement has not been opened via an OPEN statement. [FATAL]
B13	NULL CONVERSION CODE IS ILLEGAL; NO CONVERSION DONE!	A string variable that should have a value is actually null. [WARNING]
B14	BAD STACK DESCRIPTOR	This error message is generated if the lengths of the input-lists or output-lists in the CALL and SUBROUTINE statements are different, if an attempt is made to execute an external subroutine as a main program or if a file variable is used as an operand. [FATAL]
B15	ILLEGAL OPCODE: 'C'	Object code for item indicated by RUN verb contains garbage. [FATAL]
B16	NON-NUMERIC DATA WHEN NUMERIC REQUIRED; ZERO USED!	A non-numeric string was encountered when a number was required. (A value of 0 is assumed.) [WARNING]

Error No.	Error Message	Cause
B17	ARRAY SUBSCRIPT OUT-OF-RANGE	Array subscript is less than or equal to zero or exceeds the row or column number indicated by a DIM statement. [FATAL]
B18	ATTRIBUTE NUMBER LESS THEN -1 IS ILLEGAL	Attribute less than -1 specified in READV or or WRITEV statement. [FATAL]
B19	ILLEGAL PATTERN	Illegal pattern used with MATCH or MATCHES operator. [WARNING]
B20	COL1 OR COL2 USED PRIOR TO EXECUTING A FIELD STMT; ZERO USED!	COL1 or COL2 function used before FIELD function used. (A value of 0 is assumed.) [WARNING]
B21	MATREAD: NUMBER OF ATTRIBUTES EXCEEDS VECTOR SIZE	The number of attributes in the item exceeds the dimensioned size of the array; the remaining attributes are not used. [WARNING]
B24	DIVIDE BY ZERO ILLEGAL; ZERO USED!	Division by zero attempted. (A value of 0 is assumed.) [WARNING]
B25	PROGRAM 'B' HAS NOT BEEN CATALOGED	The specified external subroutine must be cataloged before appearing in a CALL statement. [FATAL]
B27	RETURN EXECUTED WITH NO GOSUB	RETURN statement executed prior to GOSUB. [FATAL]
B28	NOT ENOUGH WORK SPACE	Not enough work space assigned at LOGON to run program. [FATAL]
B29	CALLING PROGRAM MUST BE CATALOGED	An external call cannot be made unless the calling program is also cataloged. [FATAL]
B30	ARRAY SIZE MISMATCH	Array sizes in MAT Copy statement, or in CALL and SUBROUTINE statements, do not match. [FATAL]
B31	STACK OVERFLOW	The program has attempted to call too many nested subroutines. [FATAL]
B32	PAGE HEADING EXCEEDS MAXIMUM OF 1400 CHARACTERS	Page heading is too long. [FATAL]

Error No.	Error Message	Cause
B33	PRECISION DECLARED IN SUBPROGRAM 'C' IS DIFFERENT FROM THAT DECLARED IN THE MAINLINE PROGRAM	Precision must be the same between calling programs and subroutines. [FATAL]
B34	FILE VARIABLE USED WHERE STRING EXPRESSION EXPECTED	[FATAL]
B35	'M/DICT' INVALID OBJECT OF 'CLEARFILE'; IGNORED!	[WARNING]
B36	SYSTEM DICT ILLEGAL OBJECT OF 'CLEARFILE'; ABORT	[FATAL]
B41	LOCK NUMBER IS GREATER THAN 47	[FATAL]
B42	NOT ENOUGH DESCRIPTOR SPACE	[FATAL]
B209	FILE IS UPDATE PROTECTED	[WARNING]
B210	FILE IS ACCESS PROTECTED	[WARNING]

index