Educational Data Systems 2415 Windward Lane, Newport Beach, California 92660

BUSINESS BASIC

PROGRAMMING MANUAL

This manual covers the EDS Business BASIC as it is used under the IRIS operating system. Operation on an ALICE system is identical except that certain features described herein are not available under ALICE. Each statement and function of the Business BASIC language is described in detail.

For operating procedure on an EDS system, refer to the IRIS or ALICE User Reference Manual.

EDS 1016-2

TABLE OF CONTENTS

1. INTRODUCTION

2. ELEMENTS OF BUSINESS BASIC
   2.1 The BASIC Statement
   2.2 Numbers and Precision
   2.3 Simple Variables
   2.4 Arithmetic Operations
   2.5 Expressions
   2.6 Relations
   2.7 Arrays and Array Variables
   2.8 Strings and String Variables

3. FUNCTIONS
   3.1 Trigonometric Functions
   3.2 Transcendental Functions
   3.3 Mathematical Functions
   3.4 Logical Functions
   3.5 Number Manipulation Functions
   3.6 Special Functions
   3.7 User Defined Functions
   3.8 Dummy Functions

4. THE FUNDAMENTAL STATEMENTS OF BASIC
   4-1 END and STOP
   4-2 LET
   4-3 INPUT
   4-4 READ and DATA
   4-5 RESTOR
   4-6 PRINT
   4-8 PRINT USING
   4-11 GOTO
   4-12 GOSUB and RETURN
   4-13 ON
   4-14 DIM
   4-16 IF
   4-17 REM
   4-18 FOR and NEXT
   4-22 DEF
   4-23 RANDOM
   4-24 SIGNAL 1 (Send Signal)
   4-25 SIGNAL 2 (Receive Signal)
   4-26 SIGNAL 3 (Pause)
   4-26 KILL
   4-27 CALL
   4-28 CHAIN

5. MATRIX ALGEBRA
- 5-2 MAT...ZER
- 5-3 MAT...IDN
- 5-4 MAT...CON
- 5-4 MAT Assignment
- 5-5 MAT PRINT
- 5-6 MAT INPUT
- 5-7 MAT READ
- 5-8 MAT...TRN
- 5-9 MAT Add
- 5-10 MAT Subtract
- 5-11 MAT Scalar Multiply
- 5-12 MAT Multiply
- 5-13 MAT Invert
- 5-14 DET Function

6. STRING PROCESSING
- 6.1 String Expressions
- 6.2 Use of Subscripts
- 6.3 The DIM Statement
- 6.4 The LET Statement
- 6.5 The IF Statement
- 6.6 The INPUT Statement
- 6.7 The PRINT Statement
- 6.8 The LEN Function

7. DATA FILE ACCESS
- 7.1 The MRN Function
- 7.2 Notes on Locked Records
- 7.3 Notes on Open Files
- 7-4 BUILD #
- 7-6 OPEN #
- 7-6 CLOSE #
- 7-7 WRITE #
- 7-8 READ #
- 7-9 PRINT #, PRINT # USING
- 7-10 MAT WRITE #
- 7-11 MAT READ #
- 7.4 Contiguous Data Files
- 7.5 Access to a Contiguous Data File

APPENDIX I: PROGRAM EXAMPLES
APPENDIX II: TRIG FUNCTION IDENTITIES

1. INTRODUCTION

BASIC was developed at Dartmouth College in the early 60's as an easy to learn yet powerful programming language well suited to interactive use on a time sharing computer. The success of the language is indicated by the fact that over 90% of all time sharing computers in the U.S. offer BASIC. This popularity results from a number of characteristics:

A simple grammar based on a small number of English directives.

Facilities for handling strings and matrices as well as arithmetic expressions.

Built-in editing features that facilitate debugging and program modification.

Ease of translation by an interpreter. Use of an interpreter makes it possible to write and debug problems interactively. For example, a section of a program can be written and run; lines can be added, deleted, or modified; immediately, the revised program can be rerun without waiting for a compilation.

BUSINESS BASIC, the version of the language described here, is designed to preserve the characteristics which have made BASIC practically the universal time sharing language, particularly for instructional uses and scientific programming. It adds further capabilities which enhance its utility, especially for business applications. The principal extensions of Business BASIC are: extended precision decimal arithmetic, PRINT USING, data files, signalling, chaining, many special functions, and provision for large strings and arrays.

Extended precision decimal arithmetic overcomes two problems in most BASIC systems: limited precision and conversion errors. Most BASIC systems represent numbers internally in floating point binary form, typically to an accuracy of 23 or 24 bits. This results in six decimal digits of precision in which the sixth is somewhat suspect because of errors introduced through the conversion from decimal to binary and back. Business BASIC provides four precision options: 1 word integers (in the range $\pm$ 7999) and two, three and four word floating point numbers which give respectively, six, ten, and fourteen decimal digits of accuracy. Furthermore, all numbers are carried in decimal form, and the arithmetic is entirely decimal so that conversion and conversion errors are eliminated.

The second major extension is PRINT USING, which simplifies report generation by providing COBOL-like picture formats. These are used to position column headings, line up decimal points, float dollar signs, insert commas, and provide the other controls required for technical and financial reporting.

Data files provide random access data storage on the disc. Formatted files will store up to 65536 bytes of information each (over 16 million bytes in an extended IRIS file), allow random addressing of items, and perform data type checking. Contiguous files (on IRIS only) can store even more data and provide faster access but without the item type checking. A text file holds a single string of up to 65535 ASCII characters (over 16 million characters in an extended IRIS file).

Chaining allows very large programs to be segmented for execution in a system with a relatively small amount of core. Small segments also permit faster swapping for more efficient system operation. Signalling allows programs on different ports, as well as program segments, to communicate with each other. And the extended function set includes facilities for taking floating point numbers apart and putting them back together.

Business BASIC provides these facilities along with such more common features as direct execution (desk calculator mode), string processing, matrix algebra, and the CALL statement. The usefulness of the string and matrix operations is increased by the provision for large strings and arrays, limited only by the program storage available, as well as by such features as substrings, string comparisons, MAT INPUT, and matrix inversion in place.

Business BASIC is upward compatible with Dartmouth BASIC as described in Kemeny and Kurtz BASIC Programming, First Edition. Programs written in that version of BASIC will run without modification under the present system. *

* The only restriction is that all statements must be executed. The original Dartmouth translator is a compiler which permits the inclusion of certain statements which are never reached in the normal execution of the program; e.g. the DIM statement in the following:

```
10 GOTO 30
20 DIM A(30)
30 PRINT A(15)
40 END
```

This must be modified in any interpretive system, such as Business BASIC, e.g. by eliminating line 10.

This manual covers all of standard BASIC as well as the extensions of Business BASIC; however, it assumes that the reader has had some previous programming experience in a higher level language. Readers without this background may find it useful to refer to any of a number of BASIC primers, for instance, BASIC Programming by Kemeny and Kurtz. Appendix I includes several examples of programs written in Business BASIC, which illustrate many of the features of the language.

Business BASIC is extremely easy to use. A small number of system commands and control characters provide all necessary control. The programmer, at any terminal, uses these to communicate to the system that a program is to be run or listed, that a character or a line just entered is to be deleted, or that some other function is to be performed.

A user on an IRIS system should refer to the IRIS User Reference Manual for information on logging on to the system, using Business BASIC, saving programs, diagnosing and correcting errors, and use of other system facilities. Likewise, a user on an ALICE system should refer to the ALICE User Reference Manual for this information.

2. ELEMENTS OF BUSINESS BASIC

The statements of Business BASIC are constructed from a small number of elements which include: numbers, variables, arrays, arithmetic operations, functions, expressions, relations, and strings. In this chapter, each of these elements is explained to provide a basis for the description of statements in later chapters.

2.1 The BASIC Statement

A statement consists of a line number (any integer in the range 1 to 9999) followed by an English word, which is usually a directive, followed by other elements which depend upon the statement in question. Some examples are:

```
10 INPUT A,B
20 LET C=A+B
30 PRINT C
40 GOTO 10
50 END
```

A sequence of such lines, for instance the sequence shown, constitutes a program.

The line numbers serve two purposes: First, except for branching and looping, the system executes the statements in a program in the order of their line numbers. Line numbers also provide a label by which a statement may be referenced. For instance, in the example above, line 40 transfers control to line 10 so that the program is repeated over and over.

Spaces may be inserted or omitted anywhere in a statement without any effect on the execution of the program, except as shown in PRINT statements and strings (see Chapters 4 and 6).

2.2 Numbers and Precision

Numbers may be represented in any of four forms: 1-word integers, 2-word, 3-word, and 4-word floating point numbers. The precision with which a number is carried in the computer is determined by DIM and DATA statements as described in Chapter 4.

Following are the precision, number of decimal digits of accuracy, and range of the four types of number:

number type	precision	#digits	range
1-word integer	1	4	± 7999
2-word floating	2	6	$\pm .999999 \times 10^{\pm 63}$
3-word floating	3	10	$\pm .9999999999 \times 10^{\pm 63}$
4-word floating	4	14	$\pm .99999999999999 \times 10^{\pm 63}$

The decimal point is optional. Negative numbers have a "-" preceding them. Positive numbers may have a "+" preceding them, but as it is not necessary, it is usually omitted. The following are examples of numbers acceptable in Business BASIC:

```
+4
-3
 3.456789
 123456.78901234
-12345678901234
.0000000000123456789O1234
```

A number may also be typed using the "E-format", where E represents "times 10 to the power". For example,

1.526 E+6 is read "1.526 times 10 to the sixth power" and equals 1,526,000

8E-5 is read "8 times 10 to the negative fifth power" and equals .00008

All floating point numbers must be within the limits of 10^{-63} and 10^{63}. However, the computer stores floating point numbers with a maximum of six, ten or fourteen digits, depending upon the precision selected. Two additional digits store the sign and exponent of a floating point number.

If a number is entered having more than the number of significant digits specified for it, the computer will truncate the number to the selected precision and, if necessary for correct representation, express it in E form. For example, if a number which is specified to have 2-word (6 digit) precision is entered as 123,456,789, the computer will change it to 1.23456E+8.

2.3 Simple Variables

BASIC statements generally use variables to represent numeric or alphabetic data. A simple numeric variable holds one number, and consists of either a single letter (A through Z), or a letter followed by one digit. A, A1, Z, Z9 are examples.

Variables with one or two subscripts may also be expressed in Business BASIC. These consist of a letter or a letter and a digit followed by one or two subscripts enclosed in parentheses. For example, the subscripted variables normally written A_1, $B1_{10}$, $C9_{2,3}$ are represented in Business BASIC as: A(1), B1(10), and C9(2, 3). The subscripts may in general be any expressions (see section 2.5 for a description of expressions).

Array (matrix) variables, which can store arrays of numbers, and string variables, which can store sequences of alphabetic and numeric symbols, are other kinds of variables which are described in sections 2.7 and 2.8.

2.4 Arithmetic Operations

Arithmetic operations in BASIC are symbolized as follows:

OPERATION	SYMBOL	EXAMPLE
add	+	6+7=13
subtract	-	10-2=8
multiply	*	5*6=30
divide	/	20/4=5
exponentiate	↑	2↑3=8

The order in which operations are performed is determined by the normal rules of algebra:

1. All operations within parentheses are performed before any operations outside.

2. Operations within the same sets of parentheses are performed according to the precedence of the operators. From highest to lowest precedence, the operators are:

   1. ↑ Exponentiation
   2. * / Multiplication or division
   3. + - Addition or subtraction

3. Operations within the same sets of parentheses and of the same precedence are performed from left to right.

It is a good practice to enclose expressions in parentheses if one is unsure of the order in which they will be evaluated.

2.5 Expressions

An expression is any number, numeric variable, function, or combination of these combined by arithmetic operations and parentheses and nested according to the normal rules of algebra. All arithmetic operations must be included explicitely.

For example:

```
(2+4)/3
100*P
A+B↑3
5
2*(A*SIN(R/3.14159)-LOG(Q))
```

Note that expressions may include functions (see Section 3).

2.6 Relations

Relations between two expressions are symbolized as follows:

RELATION	SYMBOL	ARITHMETIC EXAMPLE
equals	=	2 = 2
does not equal	<>	4 <> 6
greater than	>	9 > 3
less than	<	3 < 9
greater than or equal to	>=	5 >= 4
less than or equal to	<=	8 <= 8

2.7 Arrays and Array Variables

Arrays provide convenient ways to organize numerical data in lists and tables. There are two forms as follows:

One Dimensional Arrays (Vectors)

A one-dimensional array, or vector, is a sequence of N numbers. The I-th number in the sequence is referred to as A(I), where A is the name of the array. The general format for referring to an element in a one-dimensional array is:

  <u>variable name</u> (<u>expression</u>)

where <u>variable name</u> is any letter or letter followed by a digit and <u>expression</u> is any expression which when evaluated identifies an element of the vector.

Two Dimensional Arrays (Matrices)

A two-dimensional array, or matrix, organizes data in rows and columns. The general format for referring to an element in a matrix is:

  <u>variable name</u> (<u>expression</u>, <u>expression</u>)

where the first expression identifies a row and the second expression identifies a column of the array.

An example will help clarify. Consider the following table of data, or matrix:

Item	Cost	Quantity Sold
1	$5.50	11
2	1.75	20
3	7.89	9
4	6.49	11

Assume that the matrix has been given the name R. R has 4 rows and 3 columns. Below are some elements in R, and their values.

R(1, 1) = 1	R(4, 1) = 4
R(1, 2) = 5.50	R(4, 2) = 6.49
R(1, 3) = 11	R(4, 3) = 11

Arrays are further explained in Chapter 5.

2.8 Strings and String Variables

A string is a sequence of one or more symbols. In Business BASIC, all printing characters on the user's terminal and all non-printing characters except those used as control functions may be used as string elements. In the computer, these are stored as ASCII codes with a one in the top bit instead of an even parity bit. An Appendix in the User Reference Manual lists the ASCII codes in order of increasing value.

Two of these codes or "bytes" are stored in each memory word, whereas one to four words are required to store the value of a numeric variable. A special string terminator symbol is used internally to mark the end of a string so that a string of N elements occupies (N+1)/2 words if N is odd,and (N+2)/2 words if N is even.

A literal string is simply a string enclosed in quotation marks; for example:

```
"ONE, TWO, THREE, TESTING"
```

Literal strings are used primarily in PRINT statements as described in Section 4. Strings may also be represented as the value of a string expression.

A string variable consists of a letter followed by a dollar sign or a letter digit combination followed by a dollar sign. For example:

```
A$, Z$, B9$, C0$
```

One or two subscripts may be used with a string variable to select a substring; each subscript may be a number or, in general, any numeric expression. For example:

```
A$(5), B1$(3, 10), C9$(I, J+1)
```

A variable with one subscript identifies the substring beginning at the element identified by the integer value of the subscript and ending with the last element of the string. A variable with two subscripts identifies the substring beginning at the element identified by the integer value of the first subscript and ending with the element identified by the integer value of the second. For example:

```
If  A$ = "ONE, TWO, THREE, TESTING"
Then A$(15) = "TESTING"
     A$(5, 13) = "TWO, THREE"
And if I = 8 and J(I) = I+1
Then A$(I+1, J(I)+4) = "THREE"
```

Strings may be constructed from smaller strings by concatenating literal strings and string variables, separated by commas. For example:

```
"ONE,", A$(5, 13), ",", A$(15)
```

is equivalent to A$. A string expression is any literal string, or string variable, or concatenation of literal strings and string variables.

The statements for manipulating strings are described in Chapter 6.

3. FUNCTIONS

Business BASIC provides many pre-defined functions for the programmer's use. These include four trigonometric functions, six additional mathematical functions, RND which produces pseudo-random numbers, DET (see MAT Invert) which gives the determinant value of a square matrix and LEN (see section 6.8) which provides the length of a string.

A function call has the general form

function (expression)

where function is a three-letter function name such as SIN, LOG, or SQR, and expression is any numeric expression. In Business BASIC, the parentheses are not required if the argument is a single variable or a positive number.

In addition to the pre-defined functions described in this chapter, the user may define his own functions. The method for creating and using such user-defined functions is described in the section on the DEF statement in Chapter 4.

3.1 Trigonometric Functions

Three trigonometric functions and one inverse trigonometric function are provided in Business BASIC. Their function names and meanings are:

SIN	sine
COS	cosine
TAN	tangent
ATN	arctangent

The argument for the sine, cosine, or tangent function is an angle expressed in radians. Although any angle will be accepted as a valid argument, some accuracy will be lost if the angle is outside the range $\pm 2\pi$ since the function routine must first reduce the angle to the first quadrant before evaluating the function. If the angle is known in degrees, it must be converted to radians before it is used as the function argument. This may be done as the function is called; for example:

```
100 LET S=SIN (A*3.1415926535898/180)
110 LET B=B+TAN (A/57.295779513084)
```

The argument of the arctangent function may be any real number (the tangent of any angle). The result will be an angle in the range $\pm \frac{\pi}{2}$ radians.

Refer to APPENDIX II for identities which may be used to calculate the other trigonometric functions in terms of the above four functions.

3.2 Transcendental Functions

Three transcendental functions are provided in Business BASIC. The names of these functions and their meanings are:

SQR	Square Root
LOG	Natural log (logarithm to base e)
EXP	Exponential (the constant e raised to the power of the argument value)

To compute the common log (log to base ten) of a number, use the identity:

$$\log_{10}X = \log_e X/\log_e 10$$

which may be expressed in a BASIC statement as follows:

```
240 LET W = LOG(X)/2.3025850929940
```

Obviously, the base 10 counterpart of EXP(X) is 10↑X, but IXR(X) may be used if X is an integer (see Section 3.5).

Random numbers are provided by the RND function. In some systems, the argument of the RND function is ignored, and only numbers in the range zero to one are generated. In Business BASIC, however, the argument may be used to specify the range over which numbers are to be generated.

An argument value of zero indicates the standard range:

$$0 \leq \text{RND}(0) < 1$$

Any other argument value x indicates the range of the result to be:

$$0 \leq \text{RND}(x) < x \quad \text{for } x \neq 0$$

The argument value may also be negative, in which case the range is between 0 and the negative argument, i.e., the random number will be negative.

The "random" numbers actually come from a sequence of pseudo-random numbers generated by the computer. Over 65,000 numbers will be generated before the sequence repeats.

To instruct the computer to "pick" a random number between 1 and N, inclusive, the following is generally used:

```
INT (RND(N))+1
```

For example, INT (RND(6))+1 will be a random integer from 1 to 6, inclusive, and may be used to simulate the throwing of one die, or used twice to simulate throwing a pair of dice.

3.3 Mathematical Functions

Five mathematical functions are provided in Business BASIC. The function names and their meanings are:

ABS	Absolute value
SGN	Algebraic sign
INT	Integer value
FRA	Fractional portion
RND	Random number

ABS(x) will yield the absolute value of x (any expression); i.e., if x is negative it will be changed to the same positive value.

SGN(x) will yield zero if the argument is zero, +1 if the argument is greater than zero, or -1 if the argument is negative. Sometimes called the signum function to distinguish it from the sine function.

INT(x) will yield the most positive integer that does not exceed the argument. For example:

```
INT (0.5) = 0
INT (3.999) = 3
INT (-4.6) = -5
```

FRA(x) will yield the fractional part of the value of the argument. For example:

FRA(2.3065) = 0.3065
FRA(5) = 0
FRA(-8.149) = -0.149

For positive values of x (any expression) FRA(x) yields the same value as x-INT(x). Note, however, that this is not true if x is negative.

3.4 Logical Functions

One logical function is currently provided in Business BASIC. It is:

NOT	Logical inversion

NOT(x) will yield one if the argument is zero, or zero if the argument is non-zero.

3.5 Number Manipulation Functions

Three functions are provided in Business BASIC for taking floating point numbers apart and putting them back together. They are:

MAN	Mantissa portion
CHR	Characteristic portion
IXR	Integer exponent of radix

A floating point number, x, may be represented as:

$x = Mr^c$

where M is the mantissa, a signed number between zero and one, r is the radix (r = 10 in EDS Business BASIC), and c is the characteristic, a signed integer which is the exponent of r. Referring to the above equation:

MAN(x) = M
CHR(x) = c, and
IXR(c) = r^c

The number x may be represented in terms of these three functions as follows:

x = MAN(x)*IXR(CHR(x))

Following are more precise definitions of the three functions:

IXR(x) yields a value equal to $r^{INT(x)}$ where r is the radix used for internal computation. In Business BASIC the radix r is ten. In most other systems r is two.

CHR (x) extracts the characteristic portion of a floating point number. CHR(x) yields an integer N such that

$$r^{N-1} \leq x < r^N$$

where r is the radix used for internal computation (see explanation of r under IXR function).

MAN(x) extracts the mantissa portion of a floating point number. MAN(x) yields a signed fraction such that

$$MAN(x) * IXR\ (CHR(x)) = x$$

Special transcendental functions may be implemented in a DEF statement by use of these special functions.

3.6 Special Functions

The SPC function is used to obtain certain types of special information such as time, port number, etc. The argument of the SPC function indicates what information is desired as follows:

function	value	
SPC(0)	CPU time used since log on (in tenth-seconds)	
SPC(1)	Connect time used since log on (in minutes)	
SPC(2)	Hours since January 1, 1973*	} REAL TIME
SPC(3)	Part of hour (in tenth-seconds)	
SPC(4)	System creation date (hours after 1-1-73)*	
SPC(5)	Your account number	
SPC(6)	Port number of your terminal	
SPC(7)	Front panel switch setting	
SPC(8)	Last BASIC error number	

*Note: the "hours since January 1, 1973" value assumes that all months have 31 days.

3.7 User Defined Functions

The user may define special functions for his own use. Up to 26 such functions may be defined, and the definitions may be changed in the course of a program run. Refer to the DEF statement in Section 4 for the procedures to define and use defined functions.

3.8 Dummy Functions

There are three dummy functions available in Business BASIC for use in defining special functions. They are:

DFV	Dummy function, variable
DFA	Dummy function, ampersand
DFP	Dummy function, percent

A dummy function is used when a certain expression appears repeatedly within the function. Use of a dummy function temporarily assigns a value to a dummy variable. The simple user-defined function (refer to Section 3.7) has one dummy variable which is assigned a value by the function call. For example:

```
120 LET A = B + FNS(C+3)
```

assigns the value of C+3 to the dummy variable and then calls the function FNS to evaluate some user-defined function of this value. If FNS had been defined by the statement:

```
10 DEF FNS(G) = (G↑2 + 3/G)↑2 - (G↑2 + 3/G)
```

the dummy variable G would receive the value of C+3. Notice that the expression G↑2 + 3/G appears twice in this definition. Therefore, this definition may be shortened by use of a dummy function. The statement

```
10 DEF FNS(G) = DFV(G↑2 + 3/G), G↑2 - G
```

is equivalent to the previous definition. The DFV (Dummy Function, Variable) causes the expression G↑2 + 3/G to be evaluated, and this value is then assigned to the dummy variable (G in the example). Evaluation of G↑2 - G then generates the value which is returned by the function. Two additional dummy functions, DFA (Dummy Function, Ampersand) and DFP (Dummy Function, Percent), may be used in a similar manner to assign values to the dummy variables & and %, respectively.

In a statement of the form

    DEF FNu(v) = . . . DFV(expression), . . .

where u and v are any letters and v is the dummy variable, the portion of the statement

    DFV(expression),

is equivalent to including the statement

    LET v = expression

within the DEF statement, where v is the same dummy variable. Similarly,

    DFA(expression)

is equivalent to

    LET & = expression

and an analogous relationship holds between DFP and the % dummy variable. The ampersand and percent sign may then be used as variables later in the same statement, definition, or nested functions, but the values of these dummy variables are not retained from one statement to the next. DFV, DFA, and DFP are the only available dummy functions. DF followed by any other letter will cause an error.

4. THE FUNDAMENTAL STATEMENTS OF BASIC

This chapter gives a detailed discussion of each statement in the "elementary" and "advanced" BASIC language (see Kemeny and Kurtz BASIC PROGRAMMING) including many extensions to these statements. Other features and extensions such as matrix algebra, string processing, the CALL statement, by which a BASIC program may call a machine language subroutine, are described in Chapters 5, 6, and 7.

Statements: END and STOP

Form:

any line number STOP

or

any line number END

or

highest line number in program END

Examples:

```
150 STOP
100 END
9999 END
```

Purpose: The END and the STOP statements terminate the execution of a program.

Remarks: The END and STOP statements have similar effects, and may be used in any portion of the program to terminate execution of the program.

Usually, the END statement is used to stop execution as the final step of a program. Most often, the STOP statement is used to terminate execution in the midst of the program.

It is not mandatory that the last statement in a program be an END statement.

The END statement will cause the simple message "READY" to be printed, while a STOP statement causes

STOP AT line number of STOP statement

to be printed.

Statement: LET

Form: <u>line number</u> LET <u>variable</u> = <u>expression</u>

or

<u>line number</u> LET <u>variable</u> (<u>expression</u>) = <u>expression</u>

or

<u>line number</u> LET <u>variable</u> (<u>expression</u>, <u>expression</u>)=<u>expression</u>

Examples:

```
10 LET P=6
20 LET R2=Q+(T/5)
30 LET A(2)=C+5
40 LET B=B+1
50 D=P+5*Q-SQR(A(Z))
```

Purpose: This statement assigns an arithmetic value to a variable. The arithmetic value normally represents the result of calculations performed by the computer.

Remarks: In a LET statement, the symbol "=" should be read as "takes the value of", not as "equals". For example,

```
LET P=6
```

should be read "Let P take the value of 6". Therefore it is possible to have

```
10 LET B=B+1
```

which means to let the new value of B take the existing value of B with one added to it.

The word LET is not required when entering an assignment statement (see example 50 above). LET will be assumed as the statement type if no type word is entered, and the word LET will be printed when the program is listed.

Statement: INPUT

Form: line number INPUT variable, variable. . .

Examples:

```
110 INPUT A, B
120 INPUT C, D4, E
130 INPUT B(2)
140 INPUT "WHAT IS YOUR NAME? "N$
150 INPUT " "J
```

Purpose: This statement informs the system that data are to be entered from the keyboard. The system will temporarily suspend the program, type a question mark, and await data to be typed in by the programmer.

Remarks: The program:

```
10 INPUT A,B
20 PRINT "THE SUM IS"; A+B
30 GOTO 10
```

will input two numbers from the keyboard, add them, print the sum, and ask for two more numbers.

To enter more than one number in response to an INPUT statement, such as in example 120, separate the numbers by commas or press the RETURN key after each number entered. When the RETURN key is pressed after entering data, the system will not return the terminal's carriage but will remain on the same line.

To terminate a program or subroutine using the INPUT statement, two methods may be applied:

1. Use ESC key - See Chapter 2.
2. Insert after the INPUT statement an IF statement so that if a given value is inserted, the program or subroutine is terminated. For example, in the second program shown above, insert

   ```
   15 IF A=0 STOP
   ```

The standard question mark prompt character may be replaced by any prompt message given in quotes at the beginning of the statement as in example 140. If there is nothing between the quotes, as in example 150, then input will be enabled with no prompt at all.

Statements: READ and DATA

Form: <u>line number</u> READ <u>variable</u>, <u>variable</u> . . .

<u>line number</u> DATA <u>constant</u>, <u>constant</u> . . .

or

<u>line number</u> DATA <u>n%</u>, <u>constant</u>, <u>constant</u> . . .

Example Program:

```
10 FOR J = 1 TO 4
20 READ Y
30 PRINT "THE SQUARE ROOT OF"; Y; "IS"; SQR(Y)
40 NEXT J
50 DATA 3%, 2, 3.7, 94.61, .0024
60 LET E=C+Z
70 PRINT E
80 DATA  12,-32.4,9999,4E-16
```

Purpose: The READ statement instructs the system to read a number from a DATA statement, and to assign the value to the specified variable.

The DATA statement is used for supplying data in a program and for specifying the precision of that data.

Remarks: As described in The DIM statement, the precision of each variable in a program is determined by the setting of a four-position "switch" when the variable is encountered in the program for the first time. This switch is initially set to two but may be changed to a new value of 1, 2, 3 or 4 by the occurrence of n% (n=1, 2, 3, 4) in a DIM statement. n% (n=1, 2, 3, or 4) may also occur immediately after the word DATA in a DATA statement, in which case the numbers in that DATA statement are read with precision n. Thus, in line 50 above, 2, 3.7, 94.61, and .0024 are read and assigned to variable Y as three-word precision numbers.

Only one % symbol may be used in each DATA statement, and it must immediately follow the word DATA. All numbers in a given DATA statement are of the same precision. The % symbol in a DATA statement does not affect the position of the "switch". Therefore, variable E, which is first encountered in line 60, will in this case be a two-word variable. If a DATA statement does not contain n%, then its data are stored with precision determined by the current setting of the "switch".

The data are read in sequence from the first to the last DATA statement and from left to right within each DATA statement. The system initially sets a pointer to the first item of data. As the READ statements request each data item, the pointer is moved to the next data item. The RESTOR statement may be used to reset the pointer.

DATA statements are not executed and may be placed anywhere in the program, though they are usually placed near the end of the program.

Items in a DATA statement must be separated by commas, but no comma should follow the last item of data.

Statement: RESTOR

Form: line number RESTOR

Program Example:

```
10 FOR I=1 TO 4
20 READ X, Y
30 INPUT Z
40 PRINT (X+Z)/Y
50 RESTOR
60 NEXT I
70 DATA 1.2, 3.14159
```

Purpose: The RESTOR statement resets the data pointer to the first item of data making it possible for the data to be re-read.

Remarks: Line 50 in the example program resets the data pointer so that the same values are read for X and Y each time through the loop.

Statement: PRINT

Form: line number PRINT list of expressions and/or literal strings

Examples:

```
10 PRINT A
140 PRINT 6*A, B, SQR(B)+C,
300 PRINT "THE ANSWER IS";R
440 PRINT "THE SUM OF";X;"AND";Y;"IS";X+Y
610 PRINT
770 PRINT E;TAB(20);"*"
990 PRINT EXP (D+SQR(X))
```

Purpose: To print text, numbers, and computation results on the user's terminal. (See also PRINT # in Section 7.)

Remarks: Example 10 above will print the current value of variable A and then cause a carriage return and line feed.

There are 75 columns or print spaces numbered zero through 74 across each line. The line is divided into five fields of 15 spaces each, starting at columns 0, 15, 30, 45 and 60. A comma in a PRINT statement causes a column tab; i.e., it causes spacing to the beginning of the next field. Therefore, statement 140 above will print the value of 6*A starting at column zero, the value of B starting at column 15, and the value of SQR(B)+C starting at column 30. The comma at the end of the statement then causes spacing to column 45 where printing will cease without a carriage return.

After printing in the fifth field, another comma would cause a carriage return, so that the sixth field will be directly under the first field, etc.

In any case, when a PRINT statement contains more than one expression, the expressions must be separated by commas or semicolons. A semi-colon causes close packing (no column tabs). Each number is printed with either a leading minus sign or space, the value, and one trailing space. Therefore, the use of semicolons will print numbers in the closest readable form.

A verbatim message may be printed by enclosing it in quotation marks as shown in example 300. Such a "literal string" is usually separated by semi-colons to prevent column tabs.

A quotation mark may be included in a literal string by use of a double apostrophe, and a carriage return may be included by use of a CTRL Z where the RETURN is desired.

Statement 440 above is a good example of the use of literal strings. If X=7 and Y=9, then statement 440 will cause the following printout:

```
THE SUM OF 7 AND 9 IS 16
```

Each printout is followed by a carriage return and line feed unless this is suppressed by either a comma or a semi-colon at the end of PRINT statement. Therefore, an empty PRINT statement, as in example 610, will cause only a carriage return and line feed.

The TAB function may be used for further control of a printout. An expression of the form

TAB (expression);

will cause spacing to the column number specified by the integer value of the expression. For instance, statement 770 above will print the value of E, space to column 20, and print an asterisk. If printing has already occurred past the specified column, no further spacing will take place. The print line is considered to be circular; i.e., columns 75, 150, 225, etc., are considered to be the same as column zero. A negative value for the tab expression will, however, cause an error.

The TAB function may be used only in a PRINT or PRINT USING statement. Any statement such as

```
100 TAB(30);"NAME"
```

or

```
200 LET A = TAB(5)
```

will cause an error.

The result from PRINT that goes into the I/O buffer is not outputted each time but is buffered up until the buffer is filled, the user is swapped out, or another BASIC statement wants to use the I/O buffer for something other than PRINT (such as INPUT). The user can always force printing with a SIGNAL 3,0 statement (see Section 4).

Statement: PRINT USING

Form: line number PRINT USING string variable; expression list

Examples:

```
 10 DIM A$(10), B$(30)
 20 LET A$="###.##"
 30 PRINT USING A$; "ANSWER="; 1.50*4
 40 PRINT USING A$; 8, 300; TAB(40);X
 50 LET B$= "+++##   $$$.###   -$#,###.##"
 60 PRINT USING B$; 7.6, 5.4, -8500
 70 PRINT USING B$(15);X;Y;Z
 80 PRINT USING B$;X; " TIMES "; Y; "="; Z
 90 LET B$= "#.###      $****#.##"
100 PRINT USING B$; 15360000; 23.469
110 PRINT USING "####.##"; X
```

Purpose: To print text, numbers, and computational results in a format specified by a string.

Remarks: String variable is a string variable (one subscript allowed) or a literal string whose value, the format string, specifies the format in which the expression list is to be printed. The format string may contain one or more format fields which control the form of printout of the numeric expressions in the expression list. It may also contain blanks (b̸) and any other characters other than format control characters. The expression list may include numeric expressions, string expressions, commas, semicolons, and TAB functions.

Printing is accomplished by starting a scan of the expression list; any string expressions are printed and tabulations are executed in response to commas, semicolons, and TAB functions until the first numeric expression is reached. Then, a scan of the format string is begun and all characters other than format control characters in the format string are printed until the first format field is reached. The value of the numeric expression is then printed in the format specified by the format field. Next are printed all non-format characters in the format string if there are no more format fields. The scan is then resumed in the expression list, printing until the next numeric expression is reached. In this way, scans of the expression list, and of the format string alternate until the expression list is exhausted.

If the end of the format string is reached before the expression list is exhausted, the scan of the format string is repeated as often as required.

Thus, lines 20 and 30 above will produce:

```
ANSWER = 6.00
```

While lines 20 and 40 will produce (if X=20.5):

```
8.00           300.00                    20.50
```

The format string may contain any of the following types of format fields:

####

For each # in the format field, a digit (0-9) or blank (b̸) is substituted. Integers are right justified with leading blanks. Signs and other non-digits are ignored. Only integers are represented; the decimal point and any fraction after it are ignored. If the datum is too large, all asterisks are printed.

###.##

A decimal point is printed where indicated. Digit positions (#) following the decimal point are filled; no blanks are left in these positions. If the fractional portion is too long, it is truncated to fit the format. Leading zeroes in the integer portion are replaced by blanks except for a single leading zero preceding a decimal point.

Signs (+, -, ++, --)

A fixed sign (+ or -) may appear as the first symbol of a format field.

Interpretation:	+ Outputs "+" if value is positive, "-" if negative.
	- Outputs "b̸" if value is positive, "-" if negative.

A floating sign (++··· or --···) appears as the first two or more symbols in the format field. Positions occupied by the second and any additional signs can be used for numeric positions in the datum and the sign is printed immediately preceding the datum.

Fixed and Floating ($)

A fixed $ appears as the first or second character in the format field, causing a $ to be printed in that character position. The $ may appear as the second character if it is preceded by a fixed sign.

A floating dollar sign ($$· · ·) consists of at least two characters beginning at either the first or second character position in the string and causing a $ to be placed in the character position immediately preceding the first digit. If the floating $ begins in the second character position, it is preceded by a fixed sign. Only one floating character (sign or $) is permitted in a given field.

Separator (,)

The separator (,) places a comma in the position indicated except where leading zeroes (blanks) occur.

Exponent Indicator (↑)

Four consecutive arrows (↑↑↑↑) indicate an exponent field and will be filled by E±nn where each n is a digit.

Asterisk (*)

The asterisk (*) specifies asterisk protection of all leading positions within the output result which would otherwise print as blanks.

In the examples above, lines 50, 60, 70 and 80 produce the following printouts (assuming X=1000, Y=9. 999, and Z=9999):

```
  + 7              $5. 400          -$8,500.00
 $1,000.00 $      9.99  $9,999.00
+1000 TIMES $9.999 =  $9,999.00
```

Lines 90 and 100 produce:

```
1. 536E+07  $***23. 46
```

Appendix I includes business application programs which further illustrate PRINT USING.

Statement: GOTO

Form: line number GOTO line number

Examples:

```
30 GOTO 10
50 GOTO 90
```

Purpose: This statement transfers the control to the specified line.

Remarks: GOTO must be followed by a line number to which the control is to be transferred; there must be a statement in the program with that line number, or an error will occur.

The statement is useful for "jumping" to another part of the program or for repeating a task indefinitely.

A GOTO should not be used to jump into the interior of a FOR-NEXT loop because a "NEXT without matching FOR" error will occur when the NEXT statement is encountered.

Statements: GOSUB and RETURN

Form:

```
line number GOSUB line number starting subroutine
     .
     .
     .
line number RETURN
```

Examples:

```
10 GOSUB 200
20 GOSUB 430
     .
     .
     .
250 RETURN
450 RETURN
```

Purpose: The GOSUB statement transfers control to the specified line number.

The RETURN statement transfers control to the statement following the GOSUB statement which originally transferred the control.

Remarks: The GOSUB and RETURN statements eliminate the need to repeat frequently used groups of statements in a program. Such a group of statements is called a subroutine.

The portion of the program to which the control is transferred must be terminated with a RETURN statement.

A RETURN statement may be used at any desired exit point in a subroutine, and there may be as many RETURN statements as needed in each subroutine.

A subroutine that has been entered with a GOSUB can itself contain a GOSUB statement. This nesting process can be carried out to 5 levels. Each RETURN is to the previous level.

A RETURN statement cannot be executed without the previous execution of a GOSUB statement.

Statement: ON

Form: line number ON expression GOTO sequence of line numbers
or
line number ON expression GOSUB sequence of line numbers

Examples:
```
60 ON J/2-5 GOTO 150,300,100,300,40
10 ON LOG(R)+1 GOTO 95,407
20 ON J GOSUB 130,140,200,210
30 ON P+1 GOSUB 190,500,650
40 ON A GOSUB 400,400,350,410,430
```

Purpose: This statement transfers control to the line number indicated by the integer value of the expression following ON.

Remarks: The expression following ON is evaluated, and the value is integerized, but not rounded. The integer is then used to select the first, second, third, etc., line number. If the value of the expression is not positive, or if it is greater than the number of line numbers listed, the GOTO or GOSUB will not be executed; control will be transferred to the next statement following the ON statement.

The subroutine given control by an ON...GOSUB statement should be exited only with a RETURN statement.

The line numbers following GOTO or GOSUB must be separated by commas.

There may be any number of line numbers listed.

To illustrate the concept, statement 20 above will transfer control to line 130, 140, 200 or 210 if the integer value of J is 1, 2, 3 or 4, respectively.

ON...GOTO is equivalent to the statement GOTO...OF implemented on some systems.

Statement: DIM

Form: line number DIM variable list

Examples:

```
10 DIM A(15)
20 DIM B2(7,8), C4(40), D$(50)
30 INPUT B
40 DIM D(2,3), 4%, G(15), H, B$(100), 3%, R
50 LET C=B+M
60 READ X,Y,Z
70 DATA 17, 344.6699802, 2
```

Purpose: The DIM statement instructs the system to reserve the correct amount of storage space for a number, an array, or a string by specifying an upper limit on the amount of space that will be required.

Remarks: Numbers and array elements may be stored in four different formats (precisions), requiring from one to four words, as described in Section 2.2. One of the purposes of DIM is to provide a means for specifying these formats. The precision of a variable is determined by the state of a four-position "switch" at the time the variable is first encountered during each run of a program; the variable remains at that precision throughout the run. The "switch" is automatically set to position two at the beginning of a run; it can be changed to any position n(n=1, 2, 3, or 4) by encountering n% in a DIM statement. All variables that are first encountered while the switch in in position 3, for example, will become three-word variables capable of carrying ten significant digits.

DIM is also used to specify the maximum number of elements that may be stored in a one- or two-dimensional array or in a string. This is accomplished by including in the DIM statement a variable name followed by one or two expressions enclosed in parentheses. See lines 10 and 20 above for example.

For a two-dimensional array, the first expression specifies the highest row number and the second expression specifies the highest column number to be used. Since an array always includes a row zero and a column zero, an array dimensioned A(3, 5) contains four rows and six columns for a total of 24 elements.

A one-dimensional array is treated as a column vector; i.e., it has only one column (column 0), and the expression specifies the highest row number. Thus, A, in line 10, can store up to 16 elements. If the value of an expression is not integral, the integer portion of the value is used. Negative dimensions are not allowed.

A one-dimensional array which is not mentioned in a DIM statement is automatically dimensioned 10 by 0. A two-dimensional array which is not mentioned in a DIM statement is automatically dimensioned 10 by 10.

The actual working size of an array may be smaller than the size to which it is dimensioned in the DIM statement. For example, in an array dimensioned 5x5, it is acceptable to use fewer than 36 elements.

The dimension of a string variable specifies the maximum number of bytes that the string can store. String variables are not automatically dimensioned so that all must appear in a DIM statement.

A DIM Statement may be placed anywhere in a program. The example program above illustrates how the DIM statement determines precision. In this program, B will be a two-word variable since it is encountered before the "switch" is changed from its initial position. Likewise, array D will be composed of two-word numbers. The 4% moves the switch so that vector G and variable H will be of four-word precision. B$ is dimensioned as a 100 character string (the use of % symbols has no effect on strings). The switch is again repositioned and R is created as a three-word variable. In line 50, C and M are also created as three-word variables since the DIM statement leaves the switch at position three, but B remains as a two-word variable. In line 60, the variables X, Y and Z will be three words each and will receive the values 17, 344.6699802 and 2, respectively.

The number of words required to hold data and variables in Business BASIC may be calculated from the following formulae:

type	number of words
simple variable	2 + precision
array	4 + (number of elements)*(precision)
string	4 + INT (dimension/2)
DATA statement	3 + (number of elements)*(precision)

The number of elements in an array dimensioned (R, C) is (R+1)*(C+1) including row zero and column zero.

Statement: IF

Form: line number IF expression relation expression statement
or
line number IF expression statement

Examples:
```
240 IF A+B=C*5 THEN 660
360 IF D*E GOTO 510
510 IF W(2,3)=R2+8 GOSUB 1140
405 IF B=7 LET N=N*2+Q
100 IF D3> 4 IF D3< M PRINT D3
```

Purpose: An IF... GOTO statement (the standard and most commonly used form of the IF statement) provides conditional branching capabilities; control will be transferred to the specified line number if the given condition is met. Extended forms of the IF statement allow branching to a subroutine or executing a given statement only if the condition is met. Any statement of Business BASIC may replace statement above.

Remarks: The words THEN and GOTO are synonymous in an IF statement. Either will be accepted but it will always be listed as GOTO.

Example 240 above will transfer control to line 660 if and only if the value of the expression A+B is equal to the value of the expression C*5. Otherwise, control will pass to the next statement in sequence following 240.

In the short form IF statement (single expression, no relation), the "condition" is met if the value of the expression is non-zero. Thus, example 360 above is identical to the statement

```
360 IF D*E<>0 GOTO 510
```

and will branch to line 510 if the value of D*E is non-zero. This is a particularly useful form since control will continue to the next statement in sequence if either D or E is zero.

Example 510 shows how a subroutine may be conditionally executed. Example 405 shows how an assignment may be made only if a condition is true. Since any statement including another IF statement may follow the condition expression, many tests can be performed simultaneously as in line 100 above. This example will print the value of D3 only if it lies in the range

$$4<D3<M.$$

Any number of IF conditions may be concatenated in this manner.

It is sometimes desirable to test two values for approximate equality since one or both may not be exact due to divisions or use of transcendental functions. In such a case, the statement may be written in the form

```
900 IF A-B<.01...
```

so that a small difference between the values of A and B will be accepted as equality.

Statement: REM

Form: line number REM any series of characters

Examples:
```
10 REM:  THIS PROGRAM ADDS NUMBERS
20 REM        ////*//
```

Purpose: The REM statement allows the insertion of a comment or remark into a program.

Remarks: REM lines are saved as part of the program. They appear when the program is listed, but they are ignored when the program is executed.

Statements: FOR and NEXT

Forms: line number FOR variable = expression TO expression

or

line number FOR variable = expression TO expression STEP expression

.
.
.

line number NEXT variable

Examples:

```
10 FOR A = 1 TO 5
20 FOR B3 = 6 TO -4 STEP -2
30 FOR M = J TO K+4 STEP B-D
    .
    .
    .
150 NEXT M
300 NEXT B3
600 NEXT A
```

Purpose: To create a program loop and cause it to be repeated a predetermined (or calculated) number of times. The variable, sometimes called the "index variable", must be the same in the FOR statement and its mating NEXT statement.

Remarks: The FOR statement assigns an initial value (the value of the first expression) to the index variable, and saves the value of the second expression as a limiting value. If the initial value does not already exceed the final value, control then passes to the statement following the FOR statement.

When a NEXT statement is encountered, the step value (assumed to be +1 unless specified by the word STEP in the FOR statement) is added to the index variable. If the result does not exceed the limit value, control is transferred to the statement following the FOR statement. If the result does exceed the limit value, control passes to the next statement in sequence following the NEXT statement.

The value of the index variable is deemed to exceed the limit value if it is more positive (for a positive step value) or more negative (for a negative step value). If the initial value in the FOR statement exceeds the limit value, a search is made for the matching NEXT statement, and control is immediately transferred to the statement following the NEXT statement without executing the statements within the loop.

Looping can be accomplished with the FOR and NEXT statements. For instance, the following two programs perform similar functions:

```
10  FOR A=B TO C STEP D
20  . . .
 .
 .
 .
100  NEXT A
110  . . .
```

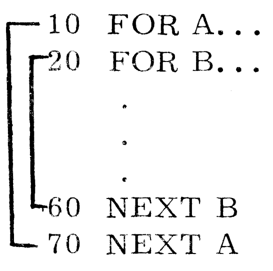
```
10  LET A=B
15  IF A>C  GOTO 110
20  . . .
 .
 .
 .
100  LET  A=A+D
105  IF  A <=C  GOTO 20
110  . . .
```

There is one important difference, however, between these two programs: changing the values of C and D within the FOR-NEXT loop will have no effect on the limit or step values since they are evaluated only once when the FOR statement is executed; changing C and D within the other program will affect lines 100 and 105.

Nesting FOR-NEXT Loops

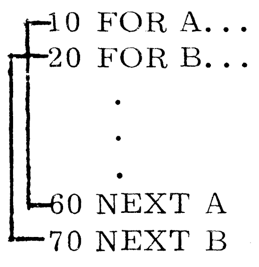
FOR-NEXT loops may be nested up to five levels deep as shown in the following examples:

legal nesting



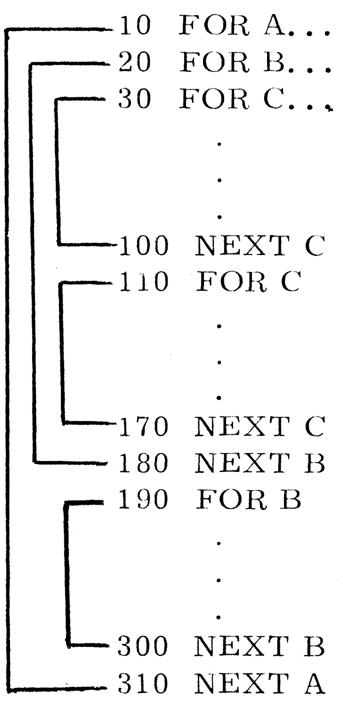
```
10  FOR A...
20  FOR B...
      .
      .
      .
60  NEXT B
70  NEXT A
```

illegal nesting



```
10 FOR A...
20 FOR B...
      .
      .
      .
60 NEXT A
70 NEXT B
```

legal nesting



```
10  FOR A...
20  FOR B...
30  FOR C...
      .
      .
      .
100  NEXT C
110  FOR C
      .
      .
      .
170  NEXT C
180  NEXT B
190  FOR B
      .
      .
      .
300  NEXT B
310  NEXT A
```

Note that for legal nesting, the mating FOR and NEXT statements can be connected without crossing lines. In the case of the illegal nesting shown, when the NEXT A statement is encountered, the system checks whether the last FOR statement encountered was a FOR A. It was not, so the index variable B loop is dropped and the FOR A statement is found. The FOR A. . .NEXT A loop is processed to completion, and an error will occur when the NEXT B statement is encountered. In certain cases, this situation may be desirable, in which case it is an acceptable programming practice as long as the circumstances which create the condition also prevent the NEXT B statement from being executed.

Another type of illegal nesting involves use of the same index variable in nested loops. For example:

```
10 FOR A...
20 FOR A...
 .
 .
 .
80 NEXT A
90 NEXT A
```

In this case the inner loop will be executed properly, but the outer loop will be lost. When a FOR statement is executed, the system checks whether an existing loop uses the same index variable, and if so, that loop and all loops nested within it are dropped. This allows programs such as the following to be properly executed:

```
 10 FOR A...
     .
     .
     .
 60 GOTO 240
     .
     .
     .
 90 NEXT A...
     .
     .
240 FOR A...
     .
     .
     .
     .
```

Special case for integer index variable:

If the index variable (D in the example below) is an integer, then the step and limit values will also be evaluated as integers. Thus, all arithmetic which must be performed by the system each time the NEXT statement is encountered will be done with integers, and it will take only one-third as long to execute the looping function. This feature should be used wherever a short FOR-NEXT loop is used and maximum speed is desired. For example:

```
10  DIM  1%, D, 3%
20  FOR D=1  TO 10 STEP 2.9
30      LET  X=5/D
40      PRINT D;X
50  NEXT D
```

will cause the following to be printed:

```
1  5
3  1.666666666
5  1
7  .7142857142
9  .5555555555
```

Note that the use of an integer (one-word precision) index variable causes the integer value of the step to be used. This is not the same as if the INT function were used, however, since the fractional part is simply ignored; a step value of -3 would be used if a step of -3.6 were specified.

Statement: DEF

Form: line number DEF FN letter (variable name) = expression

Examples:
```
10 DEF FNR(B)=2*B-C/3
20 DEF FNC(D4)=2*D4-C/3
30 DEF FNN(L)=FNC(L)+FNR(L)-1
```

Purpose: This statement allows the programmer to define his own BASIC functions.

Remarks: Up to 26 functions, FNA through FNZ, may be defined in each program.

Defined functions may be nested, as in example 30, by utilizing other defined functions within the definition. Up to 8 levels of nesting are allowed in this manner.

The DEF statement must be executed for the definition to become effective. The definition may be changed at any time by executing another DEF statement for the same function.

A defined function is used primarily when the same expression appears several places in a program. A function is defined equal to that expression, and then the function is used in the program in place of the expression. For instance, given the definition in example 20 above, the following two statements are identical in operation:

```
100 LET G=B+4*(2*Y*Z-C/3)-M
100 LET G=B+4*FNR(Y*Z)-M
```

The argument (Y*Z in the above example)of the function call (FNR) may be any expression. The expression is evaluated, and the dummy variable in the definition (D4 in example 20) is assigned that value.

The variable name in the function definition is called a dummy variable because its name is independent of all other program variables. In the above example, there might have been a variable named D4 elsewhere in the program; it would neither be affected by the above example, nor would it enter into the evaluation. Also, the dummy variable is assigned the value of the argument in the function call only for the duration of the function evaluation.

Statement: RANDOM

Form: line number RANDOM expression

Examples:

```
10 RANDOM 2
 5 RANDOM 0
```

Purpose: The RANDOM statement allows the user to exercise control over the random number sequence generated by the RND function.

Remarks: There are two problems common to most programs using random numbers:

1. The program may be difficult to debug since each run produces different results.

2. Successive runs of a debugged program may not behave independently if the "random" numbers are from a single pseudo-random sequence.

Both these problems can be resolved by use of the RANDOM statement.

The RANDOM statement is usually the first statement in a program which uses the RND function. The use of a non-zero expression, as in the first example above, will cause a certain sequence of pseudo-random numbers to be generated. Different non-zero expressions will initiate different sequences, but each RANDOM statement with the same non-zero value will initiate the same sequence.

Execution of a RANDOM statement with a zero value expression, as in the second example, causes the system clock to be used to initiate the random number sequence. Since the system clock changes each tenth second, the random number sequence which will follow a RANDOM 0 statement is unpredictable.

For best results when using the RND function, the following procedure is recommended:

1. Include a RANDOM statement with a non-zero expression at the beginning of the program while debugging.

2. Once the program is checked out, change the expression in the RANDOM statement to zero.

Statement: SIGNAL 1 (Send Signal)

Form: line number SIGNAL 1, p, x1, x2

Examples:
```
250  SIGNAL  1, B, 61, 2140
365  SIGNAL  1, D, R+1, 2*I-Q
```

Purpose: Sends a "signal", which consists of the integer values of expressions x1 and x2, to the port number given by the value of expression p. The signal will be received by the addressee only if the program running on that port executes a SIGNAL 2 (receive signal) statement.

Remarks: The expressions x1 and x2 must evaluate to positive numbers not exceeding 32767. Their integer values are then placed in a signal list along with the integer value of variable p which specifies the destination port number. In the first example above, the values 61 and 2140 are sent as a signal to port number given in B. In the second example, the values of R+1 and 2*I-Q are sent the port specified by the value of D.

The signal merely resides in the signal list until a program at the destination port executes a SIGNAL 2 statement. However, a signal will be ignored by the system if there is no user logged on at the destination port. An error will occur if the signal list is full at the time a SIGNAL 1 statement is executed, and the signal will be lost. To reduce the probability of the list being filled, any signal is automatically deleted from the list when it is about one to two hours old. Also, any signal will be deleted if the user at the destination port logs off.

Statement:	SIGNAL 2 (Receive Signal)
Form:	<u>line number</u> SIGNAL 2, p, v1, v2 or <u>line number</u> SIGNAL 2, p, v1, v2, x
Examples:	420 SIGNAL 2, P, A, B 610 SIGNAL 2, S, M(2, 3), Y, 30

Purpose: Receives any "signal" which has been sent to the port on which this statement is executed (see SIGNAL 1). The variable p will be set to the number of the port from which the signal was sent, and variables v1 and v2 will be set to the signal values (the values of x1 and x2 from the SIGNAL 1 statement). If there is no signal to be received, p will be set to minus one, and v1 and v2 will be unchanged.

It is sometimes desirable to pause and wait for a signal. In this case a time-out expression, x, may be included. The value of x specifies a delay in tenth-seconds. If a signal is received before the delay runs out, the program is immediately reactivated, and the variables p, v1 and v2 are set to the signal values. If the delay runs out first, the program is reactivated, p is set to minus one, and v1 and v2 will be unchanged.

Remarks: The variables p, v1 and v2 must be simple variables or subscripted variables; expressions are not allowed. However, x may be given as an expression. The maximum value for x is 65534 which gives a delay of nearly two hours.

In the second example above, the program will pause for three seconds (30 tenths of a second) or until a signal is received. The port number of the sender (or -1 if no signal received) will be put into variable S, and the two values of the signal will be put into M(2, 3) and Y.

A user may send a signal to his own program by pressing the BREAK key on his keyboard (use CTRL B if the BREAK key is disabled). When a SIGNAL 2 statement is executed, p will be set to the user's own port number, and v1 and v2 will both be set to zero. The SPC(6) function may be used by the program to determine its own port number.

Statement: SIGNAL 3 (Pause)

Form: line number SIGNAL 3, x

Examples:
```
660 SIGNAL 3, 100
400 SIGNAL 3, A+42
```

Purpose: Allows a program to pause (defer further execution) for a time specified by expression x in tenth-seconds. At the same time, anything buffered up in the users I/O buffer will be outputted to the channel assigned to the buffer.

Remarks: In the first example above, further program execution will be delayed for ten seconds. All inputs (except ESCAPE) and all signals will be ignored during this time. An output in progress at the time the SIGNAL 3 statement is executed will be allowed to finish. The maximum value for x is 65534, which gives a delay of nearly two hours. If X is 0, an immediate return is made to the next BASIC statement.

This statement may be used whenever it is desired to pause before executing the next statement in the program. One example of this is a program which is to loop periodically. It also will force the output of the users I/O buffer on demand.

Statement: KILL

Form: line number KILL list of string variables or literal strings

Examples:
```
540 KILL "FILE23"
200 KILL M$, "XPRL" , D$
```

Purpose: To delete disc files.

Remarks: Each literal string or string variable must contain the Filename of a disc file. The effect is the same as if the KILL command were given in the system command mode. The user's account on which the file was created will be credited for the disc blocks.

An error will occur if any of the strings given is not a legal Filename identifying a disc file that is not write protected.

If a command is given to kill a file that is open at the time on a data channel, the Filename will be removed from the INDEX immediately, but the file will remain open on the channel. The file will be deleted later when channel is closed by a CLOSE statement or cleared by program termination.

Statement: CALL

Form: line number CALL x, v1, v2, v3, . . .

Examples:
```
220 CALL 3, A, B
625 CALL 14, F2, R(2), D$
```

Purpose: The CALL statement provides a means for extending the BASIC language by adding machine language subroutines.

Remarks: The integer value of expression x selects a specific machine language subroutine. This subroutine number is assigned by the system operator in the range one to 255 when he loads the subroutine into the system.

The variables v1, v2, v3, . . . are used to pass argument values to and from the subroutine. Up to twelve such parameters may be used. Simple variables and string variables may be used either with or without subscripts, but no expressions are allowed except to select the subroutine.

For information on how to use a particular subroutine, refer to the documentation which must be provided by the user who writes the routine.

Statement: CHAIN

Form: line number CHAIN literal string or string variable

Examples:

```
400 CHAIN "RUN PART2"
310 CHAIN "BYE"
840 CHAIN M$
990 CHAIN " "
```

Purpose: The primary use of the CHAIN statement is to link together the segments of a BASIC program which is too large to be run in one piece. However, any system command can be given in a CHAIN statement.

Remarks: The CHAIN statement terminates running of the program in which it is executed and transfers control to the system, which processes the system command given in the statement. The effect is the same as if the user were to use CTRL C and then type the system command given in the CHAIN statement, but with one important exception: the user's data file channels are not closed! This allows one program segment to open a set of data files, and succeeding program segments to access the same files without requiring the Filenames. All local variables are cleared, however, and all data to be passed from one program segment to the next must be stored in data files.

Statement 400 above terminates running the current program and initiates execution of the BASIC program named PART2. If PART2 does not exist or is not a BASIC program, an error message will be printed, and the terminal will go to control mode. Any data file channels that were opened by the current program will remain open and can be referenced by PART2 without reopening them. Otherwise, the effect is the same as if there were a STOP statement in place of the CHAIN statement, and the user then pressed CTRL C and entered the command RUN PART2.

The other examples show additional uses for the CHAIN statement. The port may be automatically logged off after all calculations are finished (and the results stored in data files) by giving a BYE command as in statement 310 above. In statement 840, the string variable M$ must contain a system command. Statement 990 shows the use of an empty string in the CHAIN statement to exit to the system, similar in effect to pressing CTRL C. All channels will be cleared if the program chains to BYE or to system control mode as in examples 310 and 990 above.

5. MATRIX ALGEBRA

Business BASIC includes a set of MAT statements to facilitate fast and efficient manipulation of matrices. A matrix is defined here as an array (see Section 2.7) exclusive of row zero and column zero. Both dimensions of the array must be non-zero.

In addition to the MAT statements, Business BASIC provides for the calculation of the determinant value of a matrix which may be obtained by use of the DET function after the matrix has been inverted. For example:

```
200 MAT A = INV(B)
210 PRINT DET(X)
```

will print the determinant value of matrix B (see MAT...INV statement). The DET function of a matrix may be used as an operand at any place in an expression and at any time after inverting the matrix but before inverting another matrix. The argument of the DET function is ignored by the system, but some argument must be supplied to prevent a syntax error.

A matrix is created by the first reference to its name in a DIM or MAT statement. For example, any of these statements:

```
100 MAT INPUT D(3,4)
110 MAT READ D(3,4)
120 MAT D=ZER(3,4)
```

will create a matrix D dimensioned three rows by four columns (plus the zero row and column which are not used by the MAT statements). If the matrix already exists due to a previous MAT statement or a DIM statement, it may be given a new working size by using its name with new subscripts in a DIM or MAT statement as long as the final number of elements does not exceed the original number of elements. For example, line 130 below redimensions matrix D (defined above to be 3x4) to be 2x5:

```
130 MAT D=ZER(2,5)
```

An entire array may also be read from or written to a data file with the MAT READ# and MAT WRITE# statements. See Section 7 for details.

Statement: MAT... ZER

Form: line number MAT matrix variable = ZER

or

line number MAT matrix variable = ZER (expression)

or

line number MAT matrix variable = ZER (expression, expression)

Examples:

```
100 MAT A = ZER
200 MAT B = ZER (15)
300 MAT Z = ZER (9,14)
400 MAT L = ZER (E, F)
```

Purpose: This statement sets all the elements of the specified matrix equal to zero. The matrix may be given a new working size (as in lines 200, 300, and 400 above).

Remarks: This statement, as with other MAT statements, performs a procedure that would otherwise take several steps. For example, the statement

```
50 MAT B = ZER (R, C)
```

is equivalent to the following:

```
40 DIM B(R, C)
50 FOR I = 1 TO R
60 FOR K = 1 TO C
70 LET B (I, K) = 0
80 NEXT K
90 NEXT I
```

Statement:	MAT. . . IDN
Form:	line number MAT matrix variable = IDN or line number MAT matrix variable = IDN (expression, expression)
Examples:	210 MAT F = IDN 220 MAT G = IDN(4, 4) 230 MAT H = IDN(5, 5) 240 MAT I = IDN(B4, B4)
Purpose:	This statement establishes an identity matrix. The elements comprising the main diagonal have the value 1, and all other elements equal 0. A new working size may be specified.
Remarks:	The IDN matrix must be two-dimensional and should be "square".

In line 220 above, the matrix G has been assigned:

```
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

If the matrix is not "square", then the "main diagonal" is assumed to start at the lower right corner. For example, the statement

```
40 MAT B = IDN(3, 4)
```

will assign matrix B the value

```
0 1 0 0
0 0 1 0
0 0 0 1
```

Statement: MAT. . . CON

Form: <u>line number</u> MAT <u>matrix variable</u> = CON
or
<u>line number</u> MAT <u>matrix variable</u> = CON (<u>expression</u>)
or
<u>line number</u> MAT <u>matrix variable</u> = CON (<u>expression</u>, <u>expression</u>)

Examples:
```
150 MAT D = CON
155 MAT E = CON (8)
160 MAT Z = CON (X, Y)
```

Purpose: This statement sets all of the elements of the specified matrix equal to one. The matrix may be given a new working size (as in lines155 and 160 above).

Remarks: Otherwise it is similar to the MAT. . . ZER statement.

Statement: MAT Assignment

Form: <u>line number</u> MAT <u>matrix variable</u> = <u>matrix variable</u>

Examples:
```
225 MAT C=Y
320 MAT M2=Q6
```

Purpose: To set all of the elements of the specified matrix equal to the corresponding elements of a given matrix.

Remarks: The assigned matrix (C in example 225) is automatically dimensioned the same as the given matrix; row zero and column zero are not changed except for possible rearrangement due to rediminsioning.

Statement:	MAT PRINT
Form:	line number MAT PRINT matrix variable, matrix variable, . . . or line number MAT PRINT matrix variable; matrix variable;. . .
Examples:	50 MAT PRINT C 60 MAT PRINT C; 70 MAT PRINT A, B
Purpose:	This statement causes the system to print out one or more entire matrices, row by row.
Remarks:	A matrix may be printed in a "packed" form, where up to 12 elements may be printed on a line, by placing a semi-colon after the matrix variable; as in line 60 above. Otherwise, the matrix will be printed with five elements per row.

If the matrix variable is followed by a comma or semi-colon, an extra line feed is generated after printing the matrix to provide double spacing between matrices. More than one matrix may be printed in one statement by separating the matrix variable names by a comma or semi-colon as shown in statement 70.

Caution: If the semi-colon is used for close packing, the columns will not line up properly if any element must be printed in floating point format.

Statement: MAT INPUT

Form: line number MAT INPUT matrix variable, matrix variable, . . .

or

line number MAT INPUT matrix variable (expression), . . .

or

line number MAT INPUT matrix variable (expression, expression), . . .

Examples:
```
170 MAT INPUT F
180 MAT INPUT R(5)
190 MAT INPUT C(E, J), F
```

Purpose: This statement allows the input of an entire matrix from the terminal. The matrix may be dimensioned in the INPUT statement or given a new working size (as in lines 180 and 190 above).

Remarks: The elements of each row being entered must be separated by commas.

A complete row of data must be entered before pressing RETURN. For example, while inputing the data necessary for the execution of

```
190 MAT INPUT A(3, 4)
```

four data items must be typed in before pressing RETURN. If not enough data items are entered, the system will print a reverse slash mark and await the entire new line of data.

The matrix is filled in the following order:

1, 1; 1, 2; 1, 3; etc. across one row at a time.

Statement: MAT READ

Form: line number MAT READ matrix variable, matrix variable, . . .
or
line number MAT READ matrix variable (expression), . . .
or
line number MAT READ matrix variable (expression, expression), . . .

Examples:

```
400 MAT READ A
405 MAT READ B,C
410 MAT READ E(L,N),A
415 MAT READ F(7)
```

Purpose: This statement allows the computer to read an entire matrix from DATA statements. The matrix may be dimensioned in the READ statement or given a new working size (as in lines 410 and 415 above).

Remarks: The matrix is filled in the following order:

1, 1; 1, 2; 1, 3; etc.

i.e., a row at a time, from left to right.

Statement:	MAT. . . TRN
Form:	line number MAT matrix variable = TRN (matrix variable)
Examples:	450 MAT Q = TRN(R) 460 MAT L = TRN(A)
Purpose:	This statement establishes a matrix as the transpose of a specified matrix, i. e., the rows and columns are exchanged.
Remarks:	A sample transposition, as commanded in line 450 above, produces the following results.

```
R (original matrix)            Q(transposed matrix)

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

If the original matrix has the dimensions (M, N), the transposed matrix is dimensioned (N, M). In other words, the dimensions of the resulting matrix are opposite those of the original.

It is not necessary for the new matrix to have been previously dimensioned.

A statement of the form

```
120 MAT S = TRN(S)
```

is illegal. It may be executed, but the result will not be as expected.

Statement: MAT Add

Form: line number MAT matrix variable = matrix variable + matrix variable

Examples:
```
350 MAT A = B+C
360 MAT J = K+Q
370 MAT Y = Y+W
```

Purpose: This statement establishes a matrix equal to the sum of two matrices. The matrices must all be of the same dimensions.

Remarks: The same matrix variable may be used on both sides of the equal sign, as in line 370 above.

The addition is done element by element.

The sum matrix (matrix A in example 350) may be created by execution of this statement.

Statement: MAT Subtract

Form: line number MAT matrix variable = matrix variable - matrix variable

Examples:
```
56 MAT P = T-W
66 MAT D = F-D
```

Purpose: This statement establishes a matrix equal to the difference of two matrices. The matrices must all be of the same dimensions.

Remarks: The same matrix variable may be used on both sides of the equal sign, as in line 66 above.

The subtraction is done element by element.

The difference matrix (matrix P in example 56) may be created by execution of this statement.

Statement: MAT Scalar Multiply

Form: line number MAT matrix variable = (expression)*matrix variable

Examples:

```
200 MAT D = (4)*C
210 MAT E = (F)*Q
220 MAT R = (G/B2)*K
230 MAT M = (3)*M
```

Purpose: This statement establishes a matrix equal to the product of a numerical expression (i.e., a scalar) and a matrix.

Remarks: The same matrix variable may be used on both sides of the equal sign, as in line 230 above.

The expression by which the matrix variable is multiplied must be enclosed in parentheses.

Statement: MAT Muliply

Form: <u>line number</u> MAT <u>matrix variable</u> = <u>matrix variable</u>*<u>matrix variable</u>

Examples:
```
250 MAT L = M*N
260 MAT P = Q*R
```

Purpose: This statement establishes a matrix equal to the matrix product of two matrices.

Remarks: If the dimensions of matrix F are (A, B) and the dimensions of matrix G are (B, C), then the dimensions of the matrix produced when F is multiplied by G will be (A, C), and the resulting matrix will automatically be dimensioned accordingly.

Note that the number of columns in the first matrix must equal the number of rows in the second matrix. For example, statement 250 above is legal only if the number of columns in M equals the number of rows in N.

The matrix being assigned must not appear to the right of the equal sign. For example

```
30 MAT F = B*F
```

can not be executed.

Statement:	MAT Invert and DET function
Form:	line number MAT matrix variable = INV (matrix variable)
Examples:	250 MAT A = INV(C) 260 MAT F = INV(F)
Purpose:	This statement establishes a matrix equal to the inverse of the specified square matrix.
Remarks:	A matrix may take on the value of the inverse of its former self, as in line 260 above.

Only square, two-dimensional arrays may be used in this statement; i.e., the dimensions must be equal and non-zero.

After inverting a matrix, the DET function may be used to get its determinant value, which is evaluated as a side effect of the inversion. This determinant value is available until another matrix is inverted or a new run is initiated by the RUN command.

6. STRING PROCESSING

All printing characters on the user's terminal and most control function codes can be manipulated by the string processing extension in Business BASIC.

They are stored in the computer as ASCII codes with a one in the top bit instead of an even parity. Appendix 3 of the User Reference Manual lists the ASCII codes in order of increasing value.

Two of these codes or "bytes" are stored in each memory word, whereas one to four words are required to store the value of a numeric variable. A "string" consists of one or more bytes, and each string must be given a string variable name consisting of a letter and a dollar sign or a letter, digit and a dollar sign. For example, A$ and X4$ are valid string names. The LET, PRINT, DIM, INPUT and IF statements may be used with string variables and string expressions.

6.1 String Expressions

A string expression is defined as any combination of literal strings (symbols enclosed in quotation marks) and string variables. A string variable is defined as a letter or a letter and digit followed by a dollar sign. It may have zero, one, or two subscripts, each of which may be any numeric expression.

6.2 Use of Subscripts

A portion of a string may be manipulated by use of subscripts on the string variable. Any numeric expression may be used in subscripts; the expression is evaluated, and its integer value is used. The first subscript points to the first character to be used; the second subscript points to the last. If the second subscript is omitted or if its value is zero, then the end of the string will be the last character used.

6.3 The DIM Statement

There is no automatic dimensioning of string variables. Each string variable must be dimensioned once and only once by a DIM statement before the string name is used in other statement types. Only one dimension, the maximum number of bytes in the string, is required. Strings and numeric arrays may be dimensioned in the same DIM statement. For example:

```
10 DIM R$(25)
20 DIM D(8), A$(20), M(2,3)
30 DIM B$(10), C$(A5+D(3))
```

In statement 10, R$ is defined to be a string of not more than 25 characters. A string may be any length up to its specified dimension. Internally, all strings are given odd dimensions (by adding one of necessary). Thus, B$ in statement 30 will receive a dimension of 11.

6.4 The LET Statement

Strings may be manipulated by use of a LET statement. For example, after executing the program:

```
10 DIM A$(10), B$(15), C$(10), D$(14)
20 LET A$="ABCDE,3.56"
30 LET B$=A$(4), "XY+Z", A$(2,2)
40 LET C$=B$(2.5), "' 'M' 'X"
50 LET D$=B$,C$
60 LET E$=B$,"PDQ"
70 LET A$(6,9)="FG"
```

The string variables will have the following values:

```
A$        ABCDEFG6
B$        DE,3.56XY+ZB
C$        E,3."M"X
D$        DE,3.56XY+ZBE,
E$        (does not exist)
```

The double appostrophes within the literal string in line 40 are converted to quotation marks when actually put into C$. Likewise, a control Z within a literal string is converted to a RETURN code.

Statement 50 cannot be fully executed since the concatenation of B$ and C$ would exceed the dimension of D$. However, D$ will receive as much of the string as it can hold. Statement 60 will cause an error print out because E$ is not dimensioned. In statement 70, characters six through nine inclusive, are designated to be replaced. However, the source string "FG" is insufficient to fill the space allotted, so the remainder of A$ (from character 10 on) is shifted back to close the gap, thus leaving the string two characters shorter than before. Note that commas (except within quotes) are used only to separate fields of a string and do not cause spaces or other insertions in the resulting string.

Strings may also be converted directly into variables with the LET statement. For example, after executing the program:

```
10 DIM A$(20)
20 LET A$="123ABC4.567E+20AZ"
30 LET A=A$
40 LET B=A$(7)
50 LET C="123.456"
```

The variables A, B, and C would have the values 123, 4.567E+20, and 123.456 respectively. The statement will convert the string characters starting at the beginning of the string (or first string subscript, and stop at either the end of the string, the second string subscript, a zero byte, or an illegal character that was non-numeric or not a plus sign, minus sign, or decimal point. Note that in statement 40, the conversion stopped at the A in AZ, not the E. This is because E is a legal character in a number which has an exponent. Statement 50 shows that either string variables or literal strings may be used.

Alternatively, variables may be converted into strings with the LET statement. For example:

```
10 LET A$=12.34+C
20 LET B$(3)=(SIN3)+2.768
30 LET C$(1,14)=INT(A+.5)
50 LET A$(1,20)=12.34+C
```

In each statement above the expression to the right of the "=" sign is evaluated, converted to string characters, and then put into the destination string. If a second subscript is not given on the string dimension, then a zero-byte terminator will be stored at the end of the characters put into the string. This is the case for statements 10 and 20. In statement 30 the expression INT(A+.5) will be evaluated and then put into C$ starting at C$(2). If the result is larger than two characters, the rest will be truncated. A zero byte will not be stored because of the presence of the second subscript.

The same above statements may take the form:

<u>line number</u> LET <u>string expression</u> = <u>expression</u> USING <u>string expression</u>

This is like PRINT USING except that the variable is formatted into the string (instead of the user's I/O buffer) with the use of a format string. For example:

```
10 LET A$=12.345 USING "$$###.## "
20 LET C$(1, 4)=A+B USING "####"
30 LET B$=SIN 3 USING D$(1, 12)
```

Note that in the above examples the format string may be either literal strings or string variables. The results of the above statement would be exactly the same as in a "PRINT USING" statement. See Section 4-8 on how to use the format string characters.

6.5 The IF Statement

All of the six standard relations are allowed in an IF statement used to compare strings. Both sides of the IF relationship must be string expressions. The strings are compared byte by byte until a difference is encountered. The branch is then determined by the values of the ASCII codes where the difference first occurs. If the end of one string is reached and the other still contains valid codes, then the longer string is considered to be the greater of the two. To be equal, the two strings must be the same length.

Examples:

```
200 IF A$="YES"GOTO 350
210 IF C$,"TIME"< R$ GOTO 400
220 IF X$(N,N)>"A" LET X$(N+1,N+1)=". "
230 IF P$ PRINT A$;P$
```

Statement 230 will cause A$ and P$ to be printed if P$ is not an empty string.

6.6 The INPUT Statement

Strings may be entered during a run by use of an INPUT statement. For example:

```
100 INPUT A$(N,N)
110 INPUT X$, G$(5)
```

In statement 100, only a single character is to be accepted at character position N in A$. The rest of A$ is not affected. In statement 110, X$ may be filled up to the limit specified earlier in a DIM statement, and G$ is filled starting at character position five. Each input is terminated by a RETURN. After filling the space allotted by the program, succeeding characters up to the RETURN are ignored. If too few characters are entered, the remainder of the string is shifted back to close the gap.

6.7 The PRINT Statement

String expressions may be freely intermixed with numeric expressions in any combination in a PRINT statement. This allows considerable flexibility in formatting and labeling output data. Strings may also be used in PRINT USING. See Section 4.

6.8 The LEN Function

It is sometimes necessary to determine the length of a string. The function LEN (A$) generates a numeric value equal to the number of codes in A$. This is especially useful for extending a string when the present length of the string is unknown. For example:

```
300 LET R=LEN(A$)
400 LET A$(LEN(A$)+1)=B$(R)
410 IF LEN(A$)<20 GOTO 550
420 LET D$(LEN(D$)+1),LEN(D$)+4)=E$
430 LET Q=LEN(A$(1,4))
```

The argument of the LEN function must be a single string variable with or without subscripts.

6.9 String Arrays

Although Business BASIC does not provide for string arrays per se, the same effect can be achieved by using equal sized segments of a string as if they were elements of an array. For example, suppose an array of 15 strings of up to 48 characters each is desired. The string should be dimensioned (48+2)*15=750 characters total. (The +2 is to allow room for an end-of-string character at the end of each substring). A given element of the "array" named A$ may then be addressed as A$(N*50) where N is the number of the desired substring. Each element may be individually read and/or modified in this manner without affecting the other elements as long as care is taken not to write a substring longer than its alloted size (48 characters in the above example).

A two dimensional string array can be achieved in a similar manner. For example, if an array is to contain R rows and C columns of strings, where each string is to hold up to D characters the string should be dimensioned (D+2)*R*C characters total. A given element at row r, column c of A$ may then be addressed as A$((r*C+c)*(D+2)). An example program will help clarify the procedure:

```
100 INPUT "DIMENSIØN, #RØWS, #CØLUMNS ? "D,R,C
110 DIM A$[(D+2)*R*C],B$[D+10]
120 PRINT
130 INPUT "ENTER(1) ØR EXAMINE(2) ? "X
140 INPUT "   RØW,CØLUMN ? "R1,C1
150 IF R1<R IF C1<C GØTØ  180
160 PRINT "  NØ SUCH ELEMENT"
170 GØTØ  140
180 PRINT
190 IF X=1 GØTØ 220
200 PRINT A$[(R1*C+C1)*(D+2)]
210 GØTØ  130
220 INPUT B$
230 IF  LEN (B$)>D PRINT "    TØØ LØNG";
240 LET A$[(R1*C+C1)*(D+2)]=B$[1,D]
250 GØTØ  120
```

Lines 100 and 110 create a string (A$) of the proper size, and a secondary string (B$) for temporary storage. Lines 120 and 130 request whether the user wishes to examine or enter an array element, and line 140 requests the element's coordinates. Lines 150 through 170 are to test for legal coordinates. If entering an element, line 190 branches to the enter routine; otherwise line 200 prints the selected element. Line 220 accepts a new element, and line 230 tests whether its length is legal.

Line 240 places the first D characters of B$ into the proper position in the array.

To write such a string array into a data file it is necessary to write each substring into a separate item of the file. This may be done by use of nested FOR-NEXT loops to supply the row and column numbers. The same row and column numbers can also be used as the record and item numbers in the file. For example:

```
1000 ØPEN #1,"ARRAY"
1010 FØR R1=0 TØ R-1
1020    FØR C1=0 TØ C-1
1030       WRITE #1,R1,C1;A$[(R1*C+C1)*(D+2)]
1040    NEXT C1
1050 NEXT R1
```

This program assumes R, C, and D are defined as in the first example.

7. DATA FILE ACCESS

This chapter describes the IRIS system data file structure and access to data files from a BASIC program.

Random addressing within a formatted data file to a specific item is an important feature of EDS Business BASIC. Besides simplifying data file access, random addressing improves the efficiency of a program. For example, suppose one item in a file is to be written. In most systems the procedure would be:

(1) Read the entire record containing the item to be written,

(2) Change the value of the item, and

(3) Write the entire record back into the file.

Using Business BASIC, the procedure is:

(1) Write the item to be written.

Other efficiency improving features of the IRIS formatted data file structure include dynamic allocation of disc space as the file expands, and variable record size. A disc block is added to the file only if needed to hold a record being written by the user; there is never an empty data block. And the variable record size allows optimal use of the blocks that are needed. The user need not force his data to fit a fixed record length such as 64 words. The record length for any given formatted file may be from one to 256 words as required for the data items specified by the user. See "How to FORMAT a Data File" in the IRIS User Reference Manual for more information.

7.1 The MRN Function

The following pages give a detailed description of each statement used for data file access in the IRIS system. In addition, the MRN function (Maximum Record Number) allows the user to determine the current size of a file. The function call MRN(X) may be used at any point in a numeric expression. The integer value of expression X is used as a channel number, and the function call yields an integer one greater than the record number of the highest numbered record into which at least one item has been written in the data file which is open on that channel. An error will occur is X is not a legal channel number or if no data file is open on channel X. If the file open on the selected channel is a device file (such as $PTP) then zero will be returned as the result.

7.2 Notes on Locked Records

If a record is locked, then only the user who locked it can read or write data in that record or unlock the record. This is a necessary safeguard to prevent lost data if two or more users attempt to update a file at the same time. Suppose two users, call then Sam and Joe, decide at about the same time to modify the same item X which now has a value of twenty. Since this is a time-sharing system, the following sequence of events could occur:

1. Sam reads item X from the file and adds five to the value; he is about to write his result (25) back into X when his time slice runs out, and his program is swapped out of core.

2. Joe reads the same value of X from the file, subtracts 13, and writes his result (7) back into X.

3. Sam's program is swapped back in, and writes his result into X. The 13 subtracted by Joe has been lost.

If Sam had locked the record when he first read X, Joe would have had to wait until Sam was finished, and the correct final value for X (12) would have resulted.

To lock a record, merely omit the semi-colon at the end of the READ# or WRITE# statement. Only the user who locked the record can read or write data in that record until he unlocks it. The record will be unlocked if he:

1. Reads or writes data in the same record using a READ# or WRITE# with a semi-colon at the end of the statement,

2. Reads or writes data in any other record on the same channel, either with or without a final semi-colon on the statement,

3. Closes the channel with a CLOSE# statement, or

4. Clears all channels by terminating the RUN. The RUN may be terminated by an END or STOP statement, by an abortive error, or by pressing the ESC key.

Each user can lock only one record on each of his channels. If he desires to lock more than one record in a single file, then he may open the same file on two or more channels and lock a different record on each channel.

If a user tries to read or write data in a record that is locked by another user, his program will be swapped out, and another attempt will automatically be made during his next time slice. Since execution of a program can be delayed indefinitely in this manner, it is good practice to lock a record only if an update is intended and only for as long as necessary to perform the update.

7.3 Notes on Open Files

A data file may be opened by any number of users at the same time. The record locking features of the system make this possible without conflicts or loss of data, as explained in "Notes on Locked Records". Another type of problem that can occur is the deletion of a file by one user while another user has it open. This problem is handled automatically by the system in the following manner:

1. A user issues a legal KILL command to delete a file. The Filename is immediately removed from the INDEX so that no one can open the file after the delete command was issued.

2. If no other user already has the file open, it is immediately deleted, and the disc blocks are deallocated.

3. If another user does have the file open, he may continue to use it, but a delete flag is set in the file's header block.

4. When the last user who has the file open closes it, the delete flag is checked; the file is then deleted, and the disc blocks are deallocated.

STATEMENT:	BUILD #
Form:	line number BUILD #x, Filename expression, . . .
Example:	120 BUILD #2, "NEWFILE", #H3, N$
Purpose:	Alternative way of building and formatting a data file. Creates a new file or replaces an old file, identified by the Filename expression, on the channel specified by expression x. Several files may be created by one statement, and Filenames not preceded by a channel expression are created on successive channels.

Remarks:

In the example above, a data file with the name NEWFILE will be built on channel two, and a data file with the name given in N$ will be built on the channel number given by variable H3.

The new files will be automatically formatted by data written sequentially in record zero. Any WRITE other than to sequential items in record zero will fix the format to that already determined.

The new files will be automatically deleted by an exit from a user program or by an abortive error unless the user preserves them first by closing them with a CLOSE statement.

The item types and sizes created by a WRITE statement in a file being built will be as follows:

A string size will be INT((DIM of string)/2+1)*2 bytes if a string variable is written. A string size will be equal to the length of a substring or literal string (e.g., six bytes for A$(3, 8) or "UVWXYZ") or the next higher even number if an odd number of bytes is written.

A binary floating point number will be two words.

A decimal item will be the number size of the variable if a variable is written, of the constant if a constant is written, or four words if an expression is written.

If it is desired to build a contiguous data file instead of a formatted data file, then the Filename expression must have the number of records and the record length (separated by a ":") in brackets ahead of the actual Filename. For example:

```
120 BUILD # 3, "[200:250] FILE"
```

will build a contiguous data file called "FILE" with 200 records of 250 words each and open it on channel #3. The space required on the disc to hold the 200 records will be allocated all at once and will have sequential disc block addresses.

Statement: OPEN #

Form: line number OPEN #x,Filename expression, . . .

Examples:

```
20 OPEN #0, "PRIME", B$
45 OPEN #A, D$(6,M-1),#4, "JOE ",G$
```

Purpose: Opens an existing data file, identified by a Filename expression, on each channel identified by a channel number expression x. Additional Filenames not preceded by a channel number are opened on successive channels.

Remarks: In the first example above, the file PRIME will be opened on channel zero, and the file identified by the string in B$ will be opened on channel one. In the second example, the file identified by characters 6 through M-1 of D$ will be opened on the channel specified by the value of A, JOE5 will be opened on channel four, and the file identified by G$ will be opened on channel five.

A data file must be opened on one of the available channels, numbered zero through the maximum channel number, before it can be accessed by a READ# or a WRITE# statement. The channel must be closed by a CLOSE# statement before another file can be opened on the same channel. All channels are automatically cleared by a CTRL C system escape.

Statement: CLOSE #

Form: line number CLOSE #x1,#x2, . . .

Examples:

```
60  CLOSE #1
145 CLOSE #A-1,#4,#R
```

Purpose: Closes the file on the specified channel or channels.

Remarks: In the first example above, the file on channel one is closed. In the second example, channels A-1, four and R are closed, where A-1 and R are expressions whose values each identify a channel number, zero through the maximum channel number. An open channel must be closed before another file can be opened or built on the same channel. If a file is being built, closing it makes it accessible to other users for the first time. A file being build will be deleted if BASIC is exited by a CTRL C before closing the file.

Statement: WRITE #

Form: <u>line number</u> WRITE #c, r, i; x1, x2, x3. . . ;

or

<u>line number</u> WRITE #c, r; x1, x2, x3. . . ;

where c is a channel number expression,
r is a record number expression,
i is an item number expression, and
x1, x2, x3 . . . are numeric or string expressions.
Note: the final semi-colon is optional (see below).

Examples:

```
195 WRITE #4, 19;F, Y1+3, "JUNK", D-E
600 WRITE #C-2, 2*R, 8; 0, 2↑A, M$ (4,Q);
```

Purpose: Writes the values of the expressions x1,x2,. . . into a data file. The file to be accessed must have been previously opened on the channel specified by the channel number expression. The record number into which the data are to be written must also be specified. A starting item number may be given if desired; otherwise, item zero will be assumed.

The expressions following the semi-colon will be evaluated, and the values will be written into the specified record of the specified file, starting with the item number specified or starting with item zero if none is specified. Items not addressed in the file will not be affected.

In the first example above, item zero of record 19 of the file open on channel four will be set to the value of F, item one of the same record will be set to the value of the expression Y1+3, item two (which must be formatted as a string) will be set to the string value JUNK, and item three will be set to the value of D-E.

In the second example, the channel number and record number are given by expressions. The item number could also be given as an expression if desired. In this example, item eight of the specified record will be set to zero, item nine will be set to 2^A, and item ten will receive characters four through the value of Q of string M$.

Only sequential items of a single record may be written into by each WRITE # statement. An error will result if a variable type does not match the item type in the file.

A semi-colon should end the statement as in the second example unless it is desired to temporarily lock other users out of this record. Refer to "Notes on Locked Records" for comments on locking, unlocking, and updating records.

Statement: READ #

Form:

```
line number READ #c, r, i ; v1, v2, v3...;
                    or
line number READ # c, r ; v1, v2, v3...;
```

where c is a channel number expression,
r is a record number expression,
i is an item number expression, and
v1, v2, v3... are numeric or string variables.
Note: the final semi-colon is optional (see below).

Examples:

```
240 READ #2,6;D,W$,K(7,A-2)
415 READ #C(4)+1,R8,5;F$(4),J, J;
```

Purpose: Reads item values from a data file into the variables listed. The file to be accessed must have been previously opened on the channel specified by the channel number expression. The record number from which the data is to be read must also be specified. A starting item number may be given if desired; otherwise, item zero will be assumed.

The variables following the semi-colon will be set to the values contained in the specified record of the specified file, starting with the item number specified or starting with item zero if no item number is given. The data in the file are not affected.

In the first example above, item zero of record six of the file open on channel two is read into variable D, item one (which must be a string) is read into string variable W$, and item two is read into the element of array K at row seven, column A-2.

In the second example, the channel number and record number are given by expressions. The string variable F$ will be loaded from item five of that record starting at character position four in F$; characters one through three of F$ are not affected. Variable J will be set to the value in item six, but this value will be replaced immediately by the value of item seven. This technique may be used if the value of item six is of no interest.

Only sequential items of a single record may be read by each READ # statement. An error will result if a variable type does not match the item type in the file.

A semi-colon should be included at the end of the statement as in the second example unless it is desired to temporarily lock other users out of this record. Refer to "Notes on Locked Records" for comments on locking, unlocking, and updating records.

Note that numeric expressions are allowed in the file address (channel, record, and item numbers) and in subscripts, but an item value from the file can not be "read into" an expression.

Statement: PRINT #

Form:

```
line number PRINT #c, r, i;x1, x2, x3. . . ;
                    or
line number PRINT #c, r;x1, x2, x3. . . ;
                    or
line number PRINT #c, r, i;USING v$;x1, x2. . . ;
                    or
line number PRINT #c, r;USING v$;x1, x2. . . ;
```

Examples:

```
10 PRINT #3,21,6;254.6,D+E,F/6;
20 PRINT #C,R,I;X(1),Y(3,6),Z+W
30 PRINT #C2+1,R-1;USING B$(15);P;J;M$;
```

Purpose: To "print" text, numbers, and computational results, with or without use of a format string, to a data file or to a peripheral device.

Remarks: This statement form combines the features of the PRINT or PRINT USING statement with the facilities of the WRITE# statement. All output will be in the form of an ASCII string identical to the string that would be printed on the user's terminal if an ordinary PRINT or PRINT USING statement were used, but the string goes instead to whatever file or device is open on the specified channel. If nothing is open on the channel or if an illegal channel number is given (such as -1) then the output will default to the user's terminal. This allows the destination for all output to be selected at run time.

If printing to a formatted data file, the selected item must be an ASCII string. If printing to a peripheral device, the device must be capable of accepting an ASCII string; such devices include line printers and paper tape punches.

Note: the PRINT # statement is available only on an IRIS system.

Statement: MAT WRITE #

Form: line number MAT WRITE #c, r;v

or

line number MAT WRITE #c, r, i;v

where c is a channel number expression,
r is a record number expression,
i is an item number expression, and
v is the name of an array or string variable

Examples:

```
190 MAT WRITE #1, 20;A
200 MAT WRITE #C, 2*R, 8;B
210 MAT WRITE #3, 10, A$
220 MAT WRITE #L, R, 3;B$
```

Purpose: This statement writes data from a numeric array or from a string variable into a data file.

Remarks: The MAT WRITE # statement functions exactly as the WRITE # statement (see Page 7-7) except that an entire array or string will be written to a data file. No matrix or string subscripts are allowed. The data file address (record and item numbers) specifies only a starting position for the data transfer, and the entire array or string will be written into the file starting at that position without regard to number types, record boundaries or file format. It is the responsibility of the user's program to ensure that the data will later be read back into the same type of variable.

Statement: MAT READ #

Form: line number MAT READ #c, r;v

or

line number MAT READ #c, r, i;v

where c is a channel number expression,
r is a record number expression,
i is an item number expression, and
v is the name of an array or string variable

Examples:

```
190 MAT READ #1, 20; A
200 MAT READ #K, R+2, 8; B
210 MAT READ #3, 10; A$
220 MAT READ #1, R*2, 3; B$
```

Purpose: This statement reads data from a data file into a numeric array or into a string variable.

Remarks: The MAT READ # statement functions exactly as the READ # statement (see Page 7-8) except that an entire array or string will be read in one statement. No matrix or string subscripts are allowed. The data file address (record and item numbers) specifies only a starting position for the data transfer, and the entire array or string will be filled by copying directly from the file without regard to number types, record boundaries, or file format. It is the responsibility of the user's program to ensure that the data are read into the type of variable that matches the data form.

7.4 Contiguous Data Files

IRIS provides a second data file type, contiguous data files, which offer the experienced BASIC programmer the fastest file access possible. Any randomly selected item may be read or written in a single disc transfer. In addition, records may be of any length and any format. The format can differ from record to record as long as all records within a given file are of the same length. In return for this increased speed and flexibility, the programmer must take increased responsibility for the record formats and be careful to read data into variables of proper type.

All forms of the READ#, WRITE#, PRINT#, MAT READ#, and MAT WRITE # statements described above are valid for use with contiguous data files, but with the following differences:

1) The contiguous file is not formatted except that a record length is specified. There is, therefore, no checking by the system for a valid data type. Data are copied directly to and from the file with no number type conversion.

2) The third field in the file address is a byte displacement from the beginning of the record rather than an item number.

3) The file address specifies only a starting position for the data transfer. The transfer may continue from that position as far as desired (up to the end of the file) without regard to record boundaries.

There is also a form of the BUILD# statement for creating a contiguous file. Because there is no formatting or data type checking, it is up to the user's program to ensure that the data are read back into the same variable types as those from which they were written. Failure to take this precaution will result in unusable data. For example, if a string is written into a contiguous file and read back into a numeric variable, the numeric variable will contain characters other than digits (e.g., colons, semi-colons, etc.) The same thing will occur if, for example, three-word variables are written into the file and read back into two-word variables. Conversely, if numeric data are read into a string, incorrect characters will result, some of which cannot be printed or manipulated normally.

The contiguous file form has four important advantages over the formatted file:

1) There is no list of data block addresses in the file header and no header extender blocks. The disc address of any randomly specified record in the file is calculated automatically by the system from core-resident information, thus saving up to two disc transfers per data access.

2) There is practically no limit on the record size or on the amount of data that may be transferred in one statement.

3) Records may have different formats, determined only by the statements doing the data transfers. Records may be grouped together or linked by the program to effect variable record size.

4) The contiguous file form lends itself readily to an indexed file structure (refer to the description of the SEARCH# statement).

Refer to "How to FORMAT a Contiguous File" in the IRIS User Reference Manual for additional information.

7.5 Accessing a Contiguous File

Accessing a contiguous file is similar to accessing a formatted data file except that in a contiguous file, record numbers are used only as reference points. There is nothing to prevent a user from transferring two or more records in one command. Consider a file, FILE, which has a record length of 128 words. The program

```
10 BUILD #0, " [100:128 ] FILE"
20 DIM A$(512)
30 REM STATEMENT 40 FILLS A$ WITH A'S
40 A$="A",A$
50 WRITE #0, 0;A$
60 CLOSE #0
```

will write "A"'s into records 0 and 1 of FILE and will write one "A" and an end-of-string code into the first two character positions of record 2. It is important to note that as in formatted file, individual records may be locked; however, in the example above, only record number 0 would be locked. Records one and two would be vulnerable to simultaneous updating by another user.

Contiguous files have not items per se; however, the user may specify in the item field a byte displacement into a record. For example, if line 50 in the above program were changed to

```
50 WRITE #0, 0, 256;A$
```

it would be equivalent to

```
50 WRITE #0, 1;A$
```

except that in the first case record 0 would be locked and in the second case, record one would be locked.

APPENDIX I: PROGRAM EXAMPLES

Program 1 - Bill of Materials

```
LIST
10 DIM A$[30],B$[12]
20 PRINT "HØW MANY ITEMS"
30 INPUT N
40 FØR I=1 TØ N
45 PRINT
50 INPUT Q[I],P[I]
60 NEXT I
70 PRINT
80 PRINT "ITEM","QUANTITY","PRICE","AMØUNT"
90 PRINT
95 LET A$=" ##    ##### $###.## $#,###.##"
100 FØR I=1 TØ N
110 PRINT  USING A$;I,Q[I],P[I],Q[I]*P[I]
120 LET T=T+Q[I]*P[I]
130 NEXT I
140 PRINT
150 LET B$="$#,###.##"
160 PRINT  USING B$;"  TØTAL",,,T
170 END
```

```
RUN
HØW MANY ITEMS
? 5
? 2,750
? 25,23.50
? 10,85.35
? 145,.08
? 75,2.35
ITEM           QUANTITY       PRICE          AMØUNT

  1               2           $750.00        $1,500.00
  2              25           $ 23.50        $  587.50
  3              10           $ 85.35        $  853.50
  4             145           $  0.08        $   11.60
  5              75           $  2.35        $  176.25

  TØTAL                                      $3,128.85

READY
```

Program 2 - Payroll

```
LIST
2 DIM S$[80],T$[80]
3 LET T$="TOT GROSS=$$$$$.$$  TOT DEDUCTIONS=$$$$$.$$  TOT NET=$$$$$.$$"
4 LET S$="EMPLOYEE #  GROSS=$$$$.$$  DEDUCTIONS=$$$$.$$  NET=$$$$.$$"
5 PRINT "ENTER NUMBER OF EMPLOYES IN PAYROLL";
10 INPUT P
15 PRINT
20 LET Q=1
32 PRINT
35 PRINT "FOR EACH EMPLOYEE, ENTER: HOURS, HR. RATE & TAX RATE."
37 PRINT
40 INPUT H,R,T1
50 LET G=H*R
60 LET D=G*1E-2*T1
65 LET D=1E-2* INT (100*D+.5)
70 LET N=G-D
75 PRINT "  ";
80 PRINT  USING S$;Q;G;D;N
90 LET G1=G1+G
100 LET D1=D1+D
110 LET N1=N1+N
120 LET Q=Q+1
130 IF Q<=P GOTO 40
140 PRINT
150 PRINT  USING T$;G1;D1;N1
180 END
RUN
ENTER NUMBER OF EMPLOYES IN PAYROLL? 3

FØR EACH EMPLOYEE, ENTER: HOURS, HR. RATE & TAX RATE.

? 80,5.2,12  EMPLOYEE 1  GROSS=$416.00  DEDUCTIONS= $49.92  NET=$366.08
? 75,2.1,10  EMPLOYEE 2  GROSS=$157.50  DEDUCTIONS= $15.75  NET=$141.75
? 96,6.7,17  EMPLOYEE 3  GROSS=$643.20  DEDUCTIONS=$109.34  NET=$533.86

TØT GROSS=$1216.70  TØT DEDUCTIONS= $175.01  TØT NET=$1041.69

READY
```

Program 3 - Invoice

```
LIST
10 DIM P$[80],B$[10],Z$[7],X$[10],R[5],Q[5],T[5],U[25],F[5]
20 LET P$="+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +"
30 DIM N$[20],A$[20],S$[30],D$[60],U$[60],Y$[10]
40 LET Y$="$$$$$.$$"
50 PRINT "NAME";
60 INPUT N$
70 PRINT "MAILING ADDRESS";
80 INPUT A$
90 PRINT "CITY AND STATE";
100 INPUT S$
110 PRINT "ZIPCØDE";
120 INPUT Z$
130 PRINT "INVØICE NUMBER";
140 INPUT I1
150 PRINT "INVØICE DATE";
160 INPUT X$
170 PRINT "CUSTØMER NUMBER";
180 INPUT B$
190 PRINT "NUMBER ØF INVØICE ITEMS";
200 INPUT N
210 FØR I=1 TØ N
220   PRINT "DESCRIPTIØN ØF ITEM ";I;
230   INPUT D$[(I-1)*15]
240   PRINT "NUMBER ØF ";D$[(I-1)*15];" SØLD";
250   INPUT R[I]
260   PRINT "% DISCØUNT";
270   INPUT Q[I]
280   PRINT "UNIT CØST";
290   INPUT U[I]
300   PRINT "UNIT CØUNT";
310   INPUT U$[(I-1)*6]
320   LET T[I]=U[I]*R[I]
330   LET F[I]=T[I]*Q[I]/100
340   LET T1=T1+T[I]-F[I]
350 NEXT I
360 PRINT "";P$;"";N$; TAB (45);X$
370 PRINT A$; TAB (45);"INVØICE # ";I1
380 PRINT S$;" ";Z$; TAB (45);"CUSTØMER # ";B$
390 PRINT " QUAN.        ITEM                    % DISC.  UN.CØST      AMØUNT"
400 FØR I=1 TØ N
410   PRINT R[I]; TAB (4);U$[(I-1)*6]; TAB (12);D$[(I-1)*15];
420   PRINT  TAB (33);Q[I]; TAB (41);U[I]; TAB (49);
430   PRINT  USING Y$;T[I]-F[I]
440 NEXT I
450 PRINT "PLEASE PAY THIS AMØUNT    >>>>>>>>>>>>>>>>>>>>>";
460 PRINT  USING Y$; TAB (49);T1;"";P$
470 REM SØME ØF THE CHARACTER STRINGS IN THE ABOVE PRØGRAM HAVE
480 REM   CØNTRØL Z'S IN THEM TØ FØRCE CARRIAGE-RETURNS.
RUN
```

```
NAME? A&A AUTØ SUPPLY
MAILING ADDRESS? 44318 GLENRAVEN RØAD
CITY AND STATE? ANYTØWN, CALIF.
ZIPCØDE? 91234
INVØICE NUMBER? 3
INVØICE DATE? 10/20/72
CUSTØMER NUMBER? 67-1234
NUMBER ØF INVØICE ITEMS? 3
DESCRIPTIØN ØF ITEM  1 ? CLUTCH PLATES
NUMBER ØF CLUTCH PLATES SØLD? 12
% DISCØUNT? 5
UNIT CØST? 12.34
UNIT CØUNT? EA.
DESCRIPTIØN ØF ITEM  2 ? SPK. PLUG SETS
NUMBER ØF SPK. PLUG SETS SØLD? 24
% DISCØUNT? 5
UNIT CØST? 10.15
UNIT CØUNT? DZ.
DESCRIPTIØN ØF ITEM  3 ? MØTØR ØIL
NUMBER ØF MØTØR ØIL SØLD? 24
% DISCØUNT? 5
UNIT CØST? 12.25
UNIT CØUNT? BX.

+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
A&A AUTØ SUPPLY                                     10/20/72
44318 GLENRAVEN RØAD                                INVØICE #  3
ANYTØWN, CALIF. 91234                               CUSTØMER # 67-1234

 QUAN.        ITEM                  % DISC.  UN.CØST     AMØUNT

 12 EA.       CLUTCH PLATES            5       12.34    $140.67
 24 DZ.       SPK. PLUG SETS           5       10.15    $231.42
 24 BX.       MØTØR ØIL                5       12.25    $279.30

PLEASE PAY THIS AMØUNT   >>>>>>>>>>>>>>>>>>>>           $651.39
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +

READY
```

Program 4 - SIN/COS Plot

```
LIST
10 DIM G$[45]
20 PRINT
30 FØR X=0 TØ 6.4 STEP .2
40   LET Y1= INT (( SIN X+1)/5E-2+1.5)
50   LET Y2= INT (( CØS X+1)/5E-2+1.5)
60   FØR I=0 TØ 45
70     LET G$[I]=" "
80     IF I=21 LET G$[I]="."
90     IF I=Y1 LET G$[I]="S"
100      IF I=Y2 LET G$[I]="C"
110    NEXT I
120    PRINT G$;X
130 NEXT X
RUN

                    S                   C     0
                    .   S               C     .2
                    .       S         C       .4
                    .          S     C        .6
                    .             C           .8
                    .          C     S        1
                    .      C           S      1.2
                    .  C                S     1.4
                   C.                   S     1.6
               C    .                  S      1.8
            C       .                 S       2
        C           .               S         2.2
     C              .             S           2.4
   C                .         S               2.6
 C                  .      S                  2.8
C                   .  S                      3
C                  S.                         3.2
 C             S    .                         3.4
  C        S        .                         3.6
    C   S           .                         3.8
     S C            .                         4
   S      C         .                         4.2
 S            C     .                         4.4
S                 C .                         4.6
S                   . C                       4.8
 S                  .     C                   5
  S                 .        C                5.2
     S              .            C            5.4
       S            .               C         5.6
           S        .                 C       5.8
              S     .                  C      6
                  S .                   C     6.2
                    . S                 C     6.4

READY
```

Program 5 - Time of Day

```
LIST
10 REM  PROGRAM TO PRINT DATE & TIME-ØF-DAY
20 DIM Y$[5]
30 LET Y$="##.##"
40 DIM M$[36],A$[3]
50 LET M$="JANFEBMARAPRMAYJUNJLYAUGSEPØCTNØVDEC"
60 LET X= SPC 2
70 LET D= INT (X/24)
80 LET H= INT (X-24*D)
90 LET M= INT (D/31)
100 LET D=D-31*M
110 LET Y= INT (M/12)
120 LET M=M-12*Y
130 PRINT  TAB 22;"TØDAY IS ";M$[M*3+1,M*3+3];D+1;",";Y+1972
140 LET A$=" AM"
150 LET T=H+( SPC 3)/60000
160 IF T>=12 LET A$=" PM"
170 IF T>=13 LET T=T-12
180 PRINT  USING Y$;""; TAB 19;"THE TIME ØF DAY IS ";T;A$
190 CHAIN ""
RUN

                                                    TØDAY IS DEC 19 , 1973

                                                 THE TIME ØF DAY IS 10.42 PM
```

APPENDIX II: Trig Function Identities

The following identities may be used to compute trigonometric functions other than sine, cosine, tangent, and arctangent:

Function	Identity
contangent	$\text{ctn}(a) = \dfrac{1}{\tan(a)}$
secant	$\sec(a) = \dfrac{1}{\cos(a)}$
cosecant	$\csc(a) = \dfrac{1}{\sin(a)}$
arc sine	$\sin^{-1}(x) = \tan^{-1}\left(\dfrac{x}{\sqrt{1 - x^2}}\right)$
arc cosine	$\cos^{-1}(x) = \tan^{-1}\left(\dfrac{\sqrt{1 - x^2}}{x}\right)$
arc cotangent	$\text{ctn}^{-1}(x) = \tan^{-1}\left(\dfrac{1}{x}\right)$
arc secant	$\sec^{-1}(x) = \tan^{-1}\left(\sqrt{x^2 - 1}\right)$
arc cosecant	$\csc^{-1}(x) = \tan^{-1}\left(\dfrac{1}{\sqrt{x^2 - 1}}\right)$

For faster evaluation, calculate x^2 as x*x. For example, a user-defined function for $\sin^{-1}(x)$ could be written:

```
10  DEF FNS (X) = ATN(X/SQR(1-X*X))
```