MicroWorks Inc.

P. O. Box 44248

Cincinnati, OH 45244

MicroWorks Inc.

DOCUMENTATION

MicroWorks MicroWare Documentation

Program: Business BASIC v1.2 & 2.2
Program Authors: James Shrider, Gary Shell, William Kemmery
Documentation authors: James Shrider, Gary Shell
Date of last revision: 05/19/80

MicroWorks Inc.
P. O. Box 44248
Cincinnati, OH 45244

Table of Contents

Table of Contents

Section Page

Table of Contents

Table of Contents

CHAPTER 1

INTRODUCTION

.2 Business BASIC is an extension of .1 Business BASIC. Many new features have been added, including long variable names, additional trig functions, renumber with range, and a flashing cursor. In addition to this, many existing routines have been rewritten. The net result of this is a 30 to 35% increase in program execution speed (not including I/O time). For more information on the differences between the .1 and .2 versions of Business BASIC, see Appendix E.

This manual is not designed to teach BASIC; it is intended to be used as a reference manual. However, experimentation with the programs contained in this manual should allow the uninitiated user to become familiar with the language.

1.1 Machine Requirements

Business BASIC requires that the system in use contains either the PHIMON or DISKMON operating system. No standard Op-Sys is included in Business BASIC.

Business BASIC itself occupies 91 pages of memory (22.75k) and PHIMON/DISKMON occupies 8 pages (2k) plus 4 pages (1k) for the directory buffer. Thus, the minimum memory configuration is 32k. A standard ASCII keyboard is addressed as input port 0 and a TV-64 display driver is addressed as output port 0. Full driver routines are provided to operate a Digital Group printer which, if used, must be attached to port 3 (input and output).

Note: Business BASIC was written to run on a 2.5 Mhz machine. If your system is running at a different clock rate, certain software changes should be made. See Appendix G for details.

1.2 Keyboard Conventions

All keyboard input routines have been rewritten with a flashing cursor and auto-repeat. To use auto-repeat, simply hold down the desired key for 0.5 second and the key will begin repeating. To cease repeating, let up on the key. Note: on latched data keyboards having a one-shot strobe, auto-repeat will not work.

When BASIC is first entered, it will check the keyboard data to determine if it is latched or unlatched. It can, however, be fooled into thinking that an unlatched data keyboard is latched if a key is held down during initial entry; try to avoid this.

1.3 Manual Conventions

In this manual certain conventions have been adopted. The term <constant> refers to a numeric constant, ie., 1, 29, 52736, etc. The term <variable> refers to a numeric variable name, i.e., ALPHA.COUNTER, B, Z9 etc. The term <expr> refers to a numeric constant or a numeric variable name or a valid arithmetic expression, i.e., A+1, A+T5, etc. The term <str constant> refers to an alphanumeric constant, i.e., "TOP", "UNIT2", "345", etc. The term <str variable> refers to an alphanumeric variable name, i.e., A$, BETA8$, R7$, etc. The term <str expr> refers to an alphanumeric constant or an alphanumeric variable name or a valid alphanumeric expression, i.e., A$+B$, R7$+"X", etc. Brackets [] surrounding a parameter indicate that it is optional.

CHAPTER 2

USAGE OF BUSINESS BASIC

2.1 Entering a Program

Each program line is preceeded by a statement number used to identify the line. The valid range for these numbers is zero thru 65535. Business BASIC assumes that any line beginning with a line number is to be processed by the program editor; all others are considered commands or direct execution statements. The program editor serves as the user's method of entering and altering a program. If the statement number is valid and the line contains at least one character beyond the statement number, the line is added to the program. If the statement number is equal to an existing statement number, the existing line is replaced by the new line. If there are no characters beyond the statement number, the line with a matching statement number is deleted. An error will be generated if the statement number is invalid, if the line contains too many numeric constants, or if memory becomes full.

Blanks preceeding a statement number are ignored. The first non-numeric character delineates the statement number from the rest of the line. All blanks are ignored, except those contained in a quoted literal. Blanks are inserted automatically in listings for ease of reading. FOR-NEXT loops are indented for the same reason (see LIST command).

2.2 Making Corrections

Before typing a carriage return, corrections may be made to a line in two ways. First, typing a RUBOUT (left arrow or ASCII DEL on some keyboards) causes Business BASIC to backspace one character. Typing a CTRL-X (ASCII CANCEL) causes Business BASIC to ignore the entire line.

2.3 Line Editor

The line editor is used to change parts of a statement without having to retype the entire line. The editor is invoked by typing a dash followed by the number of the desired line.

Example: -10
(Would be used to edit line 10. If there were a line 10, the line would be listed on the screen with a flashing cursor at the beginning of the line; however, if line 10 did not exist, an

error would be reported).

Once in the edit mode, the typing the following keys will result in the given action:

```
     control H - Move cursor left.
     control I - Move cursor right.
     control J - Move cursor to end of line.
     control X - Exit edit mode and cancel line (does not alter
                 statement).
           RUB - Delete one character.
        return - Exit edit mode.  Input is taken as all
                 characters in the line to the left of the
                 cursor.
           ESC - Exit to the resident opsys.
```

Any other character will simply be inserted into the line at the point indicated by the cursor.

The editor will operate on any line up to 131 characters long. Editing a longer line is legal, but will result in the excess characters being truncated.

2.4 Multiple Statements on a Line

Multiple statements on one line are allowed. The statements should be seperated by a colon.

Example: `10 PRINT A : GOTO 40`

A statement number may occur only at the begining of a line.

2.5 Direct Execution

All statements (except GOSUB) may be entered without a statement number for immediate execution. This allows the user to examine values and perform other operations without having to run a program. Multiple statements per line may also be entered for direct execution.

2.6 Interrupting Execution

To stop a listing or execution of a program, type a CTRL-C (ASCII ETX). If a list is in progress, the output will stop at the end of the line currently being listed. If program execution is in progress, it will terminate at the end of the current statement being executed. Typing CONT (see "Commands") will resume execution of the program. Note: this feature may be disabled, if necessary; for more information, see the BRK function.

2.7 Pausing During Output

At any time when output is being sent to the TVC-64, pressing a CTRL-A will cause Business BASIC to pause until the key is let up. (This is only true for keyboards with unlatched output. On keyboards with latched output, typing a CTRL-A will cause BASIC to pause until another key is typed). This is useful for pausing during program executions and listings.

2.8 Exiting BASIC

To exit Business BASIC and enter PHIMON or DISKMON, type ESC during any keyboard input (including KEYIN). Note: this feature may be disabled, if necessary; for more information, see the BRK function.

At any other time, pressing the system reset button will cause an exit to the resident operating system.

CHAPTER 3

COMMANDS

There are two types of directives the user may present to Business BASIC: commands and statements. Commands are defined as those directives which act upon the program contained in memory. Statements, on the other hand, are defined as the individual lines of code which make up the program contained in memory. The following are the commands available in Business BASIC:

3.1 APP Command

Format: APP[#<expr>,]["<program name>"]

Used to append the specified program on drive #<expr> to the end of the program in memory. If <expr> is not specified, the default value of zero is used. If <program name> is not specified, the name in NAME will be used. Note: the line number of the appended program must be greater than the last line number of the resident program, or a "Sequence number overflow/overlap error" will be reported and the command aborted.

Example: APP#1,"PROG2"
(Append the progam "PROG2.BA" on drive one to the resident program.)

3.2 AUTO Command

Format: AUTO [<constant 1>[,<constant 2>]]

This command allows automatic numbering of statements as they are being entered into the system. <Constant 1> represents the starting statement number. <Constant 2> represents the increment to be used between statement numbers. If <constant 1> or <constant 2> is omitted a default value of 10 is assumed. To exit from the automatic numbering mode, type a CTRL-D (ASCII EOT).

Example: AUTO 20,5
(Begins auto statement numbering at line 20 and indicates an increment of 5 between numbers, i.e., 25, 30, 35 etc.)

3.3 CLEAR Command

Format: CLEAR

This command clears the contents of all variables. All dimensioned variables are deleted and all function definitions are removed.

3.4 CONT Command

Format: CONT

This command continues execution of a Business BASIC program after execution of a STOP statement or a CTRL-C interruption.

3.5 DEL Command

Formats: DEL <constant 1>,<constant 2>
DEL[#<expr>,]["<program name>"]

In the first form, the DEL command is used to delete a range of lines from the resident program.

<Constant 1> is the first line which is to be deleted; <constant 2> is the last line. Both starting and ending line numbers must be specified and must exist in the program.

Example: DEL 20,60
(Would delete lines 20 through 60).

Note: Using the DEL command causes the variable table to be cleared.

In the second form, the DEL command is used to delete the specified BASIC program from drive #<expr>. If no <expr> is specified, then drive zero will be used. If no <program name> is given, the name in NAME will be used.

Example: DEL "PROG1"
(Would delete the progam "PROG1.BA" from drive zero.)

3.6 LIST Command

Format: LIST [!][<constant 1>[,[<constant 2>]]]

This command is used to list the statements comprising the program currently in memory. There are four variations of this command, each performing a different function. The first variation is LIST with no parameters; this will list the entire program starting with the first statement. The second variation is LIST with <constant 1>; this will list the statement which has a statement number equal to <constant 1>. The third variation is LIST with <constant 1> followed by a comma, which will list the program starting with the statement number equal to <constant 1> through the last statement. The final variation is LIST with <constant 1> and <constant 2> separated by a comma; this will list all statements from the statement number equal to <constant 1> to the statement number equal to <constant 2>, inclusive.

If the "!" clause is specified, multiple statements per line will be listed on separate lines. IF-THEN-ELSE statements will also be listed in paragraph form on multiple lines. Any line reference to a line which is a REMark will have the REMark listed in brackets [] following the reference. Any line reference to a non-existant line will have the message "[UNDEFINED]" listed following the reference.

Before the actual listing begins, the contents of NAME and DATE will be output as a header.

Examples:

```
LIST
PROG1    05/01/80

10  GOSUB 100 : STOP
100  'MAIN PROGRAM
110  IF X=1 THEN #"TRUE" : RETURN ELSE #"FALSE"
   : IF Z THEN #"DEFAULT"
120  RETURN
200  GOTO 210


LIST!
PROG1    05/01/80

   10  GOSUB 100 [MAIN PROGRAM]
       STOP
  100  'MAIN PROGRAM
  110  IF X=1
          THEN #"TRUE"
               RETURN
          ELSE #"FALSE"
               IF Z
                  THEN #"DEFAULT"
  120  RETURN
  200  GOTO 210 [UNDEFINED]
```

3.7 LOAD Command

Format: LOAD [#<expr>,][“<program name>”]

This command will cause the previously saved BASIC program having the name specified by the <program name> to be loaded into memory from drive #<expr>. If <expr> is not specified, a default value of zero is assumed. If <program name> is not specified, the name in NAME will be used. Any program existing in memory will be cleared. Note: The LOAD command is also an executable statement.

Example: LOAD "PROG1"
(Would load the program "PROG1.BA" into memory from drive zero).

3.8 NAME Command

Format: NAME"<program name>"

Assigns a name to the program currently in memory. The name is from one to six characters. The extension ".BA" is implied; it need not be entered. The entered name is stored in a reserved variable called "NAME", similar in usage to the DATE variable.

Example: NAME "PROG1"
(Would load the string "PROG1" into NAME.)

Note that in order for a command (such as SAVE) to use the name in NAME, name must not have a zero length or a "No program name error" will occur.

3.9 REN Command

Format: REN [<constant 1>[,<constant 2>[,<constant 3>[,
<constant 4>]]]]

This command is used to renumber all or part of the program. All line references are also renumbered accordingly. Any reference to a non-existant line is left unchanged. <Constant 1> is used to indicate the first statement number in the resulting renumbered program. <Constant 2> is used to indicate the increment between statements in the resulting program. If <constant 1> or <constant 2> is omitted, a default value of ten is assumed. <Constant 3> is used to specify the first line that is to be renumbered. If unspecified, the default value is zero. <Constant 4> is used to indicate the last line that is to be renumbered. If unspecified, a default value of 65535 is used.

Examples: REN
(Renumbers the entire program, starting with 10 and incrementing by 10).
REN 5
(Renumbers the entire program, starting with 5 and incrementing by 10).
REN 5,2
(Renumbers the entire program, starting with 5 and incrementing by 2).
REN 5,2,20
(Renumbers line 20 of the program, starting with 5 and incrementing by 2).
REN 5,2,20,
(Renumbers lines 20 through 65535 of the program, starting with 5 and incrementing by 2).
REN 5,2,20,40
(Renumbers lines 20 through 40 of the program, starting with 5 and incrementing by 2).

3.10 RUN Command

Formats: RUN [<constant>]
RUN[!][#<expr>,]<str expr>

In the first form, this command instructs Business BASIC to clear the contents of all variables and begin execution of the program. The <constant> is an optional statement number at which to begin execution. If the constant is omitted, execution will begin with the first statement in the program.

In the second form, this command will cause the previously saved BASIC program having the name specified by the <str expr> to be loaded into memory from drive #<expr> and executed. If <expr> is not specified, a default value of zero is assumed. Any program existing in memory will be cleared. If the exclamation point clause is specified, the contents of any existing variables will remain unaltered (however, all user defined functions will be cleared); otherwise, all variables will be cleared. Note: The RUN command is also an executable statement.

Example: 100 RUN!#1,"PROG1"
(Would load the program "PROG1.BA" into memory from drive one and execute it without altering any existing variables).

3.11 SAVE Command

Format: SAVE[#<expr>,]["<program name>"]

This command will cause the program in memory to be saved on drive #<expr> under the name specified by <program name>. A ".BA" extension will automatically be added to the <program name>. If the <expr> is not specified, a default value of zero is assumed. If <program name> is not specified, the name in NAME will be used. This command will not affect the program in memory or the contents of any variables, including NAME. Note: The SAVE command is also an executable statement.

Example: SAVE "PROG1"
(Would save the program in memory on drive zero under the name "PROG1.BA").

3.12 SCR Command

Format: SCR

This command is used to clear the program and data storage area in Business BASIC so that a new program may be entered. This command will also clear NAME.

3.13 XREF Command

Format: XREF<search string>

This command will search for and print the number of any line containing an occurrance of the specified string. This may be used to cross reference a variable, line number, keyword, and just about anything else.

Examples: XREF A1
(Would print the numbers of all lines in which A1 is used).
XREF GET
(Would print the numbers of all lines in which the GET statement is used).
XREF 100
(Would print the numbers of all lines which reference line 100).
XREF A(
(Would print the numbers of all lines in which the array A is used).

CHAPTER 4

ELEMENTS OF THE BASIC LANGUAGE

4.1 Constants

Storage of all numeric constants and all other numeric items is in Binary-Coded-Decimal (BCD). Eight digits of precision are maintained at all times; therefore, rounding off may occur within a program if necessary. The magnitude range of constants is .1E-63 thru .99999999E+63.

4.2 Variable Names

Simple numeric variable names consist of a letter followed optionally by up to 31 letters, digits, and periods e.g., A9, R, LOOP.COUNT83). All characters of a variable name are significant (e.g., "ALPHA.TEST" and "ALPHA.TEST.COUNT" are two distinctly different variables. The only restriction on names is that they may not contain BASIC keywords (e.g., "WORD" is not a legal variable because it contains the keyword "OR". See Appendix F for a complete list of BASIC keywords). Each numeric variable occupies a number of bytes of memory equal to the length of the name plus seven. Once assigned a value these bytes cannot be released without issuing a CLEAR or SCR command.

Numeric array variable names take the same form as simple numeric variable names. The number of bytes required by a numeric array can be calculated by the following formula:

> Number of bytes = (Total number of elements * 5) + (Number of dimensions * 2) + (length of name) + 5

Unlike simple numeric variables, the memory occupied by an array can be released by the program (see "UNDIM").

String variable names take the same format as simple numeric variables with the addition of a "$" (e.g., ALPHA$, R0$, Z9$, etc.) The memory requirements for a string variable can be calculated by the following formula:

> Number of bytes = (Dimension size of string) + (Length of name + 7

Memory occupied by strings may be released in the same way as memory occupied by arrays. There are no string arrays supported by Business BASIC.

Numeric user defined function names (see "User Defined Functions") are in the same format as simple numeric variable names preceeded by "FN" (e.g., FNALPHA, FNR0, FNZ9, etc.)

String user defined function names are in the same format as string variable names peceeded by "FN" (e.g., FNALPHA$, FNR0$, FNZ9$, etc.)

Note that the same variable name can be used to refer to a simple numeric variable, a numeric array, a string, and a function definition. As an example the names A0, A0(0,0), A0$, FNA0, and FNA0$ all refer to different items and no relationship between these items is assumed to exist.

DATE is a special variable name in Business BASIC, used to contain the current date. DATE is used in statements as a string name; however, no substring refrence is allowed. If DATE is used in an INPUT statement an automatic prompt, "Enter date: <MM/DD/YY>", will be generated.

NAME is also a special variable name, used to contain the current program name. NAME is used in statements as a string name; however, no substring reference is allowed. NAME is set whenever a new program is loaded into memory, or when the NAME command is used.

4.3 Arrays

Arrays in Business BASIC may be dimensioned with any number of dimensions. Indexing of arrays is zero relative, i.e., the first element of an array is element zero. An array established by the statement 10 DIM X(2) would contain 3 elements: X(0), X(1) and X(2). Care must be taken so that an array is not re-dimensioned without first having been released (see "UNDIM").

4.4 Strings

Strings in Business BASIC may be dimensioned to any size (with the obvious upper limit of available memory). Strings may be referenced in two ways. The first is to reference the entire string. This is done by using the string name alone (e.g., 20 LET B$=A$ ... this says "Copy the contents of the entire string called A$ into the entire string called B$"). The second is to reference part of the string (hereafter called a substring); this is done by using the string name followed by two parameters enclosed in parentheses. The first parameter indicates the starting position in the string and the second indicates ending position (e.g., 20 LET B$=A$(4,7) ... this says "Copy the contents of the fourth thru the seventh character of the string called A$ into the entire string called B$"). If only the first parameter is provided, the substring is assumed to begin with the indicated position and continue thru the end of the string (e.g., 20 LET B$=A$(4) ... this says "Copy the contents of the fourth thru last character of the string called A$ into the entire string called B$").

Combining two strings into one (i.e., concatenation) is acomplished by use of the plus sign (e.g., 20 LET B$=A$+C$ ... this says "Append the contents of the entire string called C$ to the contents of the entire string called A$ and place the result in the entire string called B$. The strings A$ and C$ are unaffected").

All strings should be initialized before accessing them using substring notation.

Example:
```
10  DIM A$(10)
20  FOR I=1 TO 10
30  A$=A$+" "
40  NEXT I
50  B$=A$(1,5)
```

When assigning a value to a string, if the result string is not subscripted the result string is replaced by the assigned value.

Example:
```
A$="ABCDEF"
PRINT A$
ABCDEF
```

If the result string is subscripted and the value is longer than the result string, the assigned value will be truncated to fit. Further, if the assigned value is shorter than the result substring, the extra characters in the result substring will be unaffected.

Example:
```
A$="ABCDEF"
PRINT A$
ABCDEF
A$(2,3)="WXYZ"
PRINT A$
AWXDEF
A$(2,3)="R"
PRINT A$
ARXDEF
```

When using string IF statements, strings of unequal length will not be considered equal. If two strings of different lengths are equal up to the length of the shortest string, the shorter string is assumed to be the lesser in value. Strings in DATA statements and string constants must be enclosed in quotes.

```
Example:  10  LET A$="ABC"
          90  DATA "XYZ","UNIT"
```

4.5 Operators

The valid numeric operators for Business BASIC are: "+" for addition, "-" for subtraction, "*" for multiplication, "/" for division, "↑" for exponentiation, MIN (takes the lesser of two values), MAX (takes the greater of two values), and MOD (takes the remainder resulting when the first value is divided by the second). When an arithmetic operation is executed, any exponentiation in the statement is done first. Next, any multiplication or division is executed in left to right order. Then, any addition or subtraction is executed, also in left to right order. Finally, any MIN, MAX, or MOD operations are performed in left to right order.

```
Examples:  PRINT 1+2*5+4
            15
           PRINT 2 MIN 1+2
            2
```

If execution in some other order is needed, parentheses may be used. If a pair of parentheses appears within another pair, execution starts with the innermost pair and moves outward.

```
Example:  PRINT (1+2)*(5+4)
           27
          PRINT (1+2)*(5+4)+4*2
           35
          PRINT (1+2)*((5+4)+4*2)
           51

          PRINT 2*(11 MOD 4)
           6
```

Relational operators are "=" for equal to, "<" for less than, ">" for greater than, "<>" for not equal to, "<=" or "=<" for less than or equal to, and ">=" or "=>" for greater than or equal to. A relational operation assigns a value of one for true and zero for false.

```
Example:  PRINT 2=3
           0
          PRINT 2<3
           1
          A$="Test"
          PRINT (A$="Test")
           1
```

Boolean operators are "AND", "OR", and "NOT". Boolean operands are considered true if not equal to zero and false if equal to zero. Results of boolean operations are one or zero.

```
     Example:  PRINT 1 AND 0
                0
               PRINT 3 OR 0
                1
```

(Note: The usual usage for the boolean operators is for complex IF statements, but they can be used as above for other purposes such as program analysis of complex logic structures).

The following is a chart describing the execution order of precedence:

```
Highest    NOT,unary -
           ↑
           *,/
           +,-
           MOD,MIN,MAX
           <,<=,<>,=,=<,=>,>,>=
Lowest     AND,OR
```

CHAPTER 5

STATEMENTS

The statements available in Business BASIC can be broken down into five categories: data handling, program control, general input/output, file handling, and miscellaneous statements.

5.1 Data Handling Statements

5.1.1 CONVERT Statement

```
Formats:  10  CONVERT <expr> TO <str variable> (<mask>)
          10  CONVERT <str variable> TO <variable>
```

The CONVERT statement is used to convert numbers to strings and strings to numbers. The mask is used to indicate the format of the resulting conversion. The mask is made up of the following characters:

Character	Function
-	optional minus sign
#	character that will be replaced by a digit
.	optional decimal point

```
Examples:  LET X=234.56783
           CONVERT X TO A$(###.##)
           PRINT A$
           234.57
           CONVERT X TO A$ (-###.##)
           PRINT A$
           0234.57
           X=-234.56783
           CONVERT X TO A$ (###.##)
           FORMAT ERROR
           CONVERT X TO A$ (-###.###)
           PRINT A$
           -234.568
```

5.1.2 DELAY Statement

```
Format:  10  DELAY [<expr>]
```

This statement causes a delay to be generated between characters on output. The delay generated is <expr> times .00266 seconds, where the expr is 0 to 255. (Note: If <expr> is omitted the default value 0 is assumed).

Example: 10 DELAY 100
(Causes a .266 second delay between characters on output).

5.1.3 DIM Statement

Format: 10 DIM <variable name>(<expr 1>[,<expr 2>...[,<expr n>]])[,<str variable name 2>(<expr 1>)]

The DIM statement is used to establish the maximum size requirements for strings or numeric arrays. There is no limit to the number of dimensions attributed to a numeric array. A string variable may have only a single dimension. If no DIM exists for a numeric array, it is assumed to be a single dimensional array of ten elements. Non-dimensioned strings are assumed to have a maximum length of ten characters.

Example: 10 DIM X(2,4,5,2,3),CAR$(23)

5.1.4 FILL Statement

Format: 10 FILL <expr 1>,<expr 2>[,<expr 3>[...,<expr n>]]

The FILL statement is used to place one or more bytes into specific consecutive locations in memory. The value of <expr 1> is used as the address of the first byte. <Expr 1> may have a value from 0 to 65535. <Expr 2> may be either numeric (with a range of 0 to 255) or string. String expressions will be stored on a byte-by-byte basis, from first to last.

Example: 10 FILL 65534,255,"T"
(Places the hex value FF at hex address FFFE, and the hex value 54 at hex address FFFF).

5.1.5 HEX Statement

Format: 10 <variable> = HEX<str expr>

The HEX statement will take <str expr> and assume that the data contained therein is pairs of hexadecimal digits. These digits will be converted into true hexadecimal and placed in the variable named. If the variable is numeric, two hex pairs are converted. If the variable is a string, any number of hex pairs may be converted.

Example:
```
A$=HEX "D4C5D3D4C9CEC7"
PRINT A$
TESTING
```

```
X=HEX"D875"
PRINT X
 55413
```

5.1.6 LET Statement

```
Formats:  10  [LET]<variable>=<expr>
          20  [LET]<str variable>=<str expr>
```

This statement is used to assign the value of <expr> to the <variable>. The equal sign should be read as "is replaced by". Thus, the first example below would be read "let A be replaced by the value X plus one".

```
Example:  10  LET A=X+1
          20  D$="test"
```

(Note: Multiple assignments such as "10 A=B=0" are not allowed. The proper form would be "10 A=0 : B=0").

5.1.7 LINE Statement

```
Format:  10  LINE <expr>
```

This statement establishes the maximum line length of the system's terminal device. No input or output line longer than the specified number of characters will be allowed. If an attempt is made to use a longer line on input, all characters except carriage return, RUB, and CTRL-X will be ignored. On output a new line will automatically be generated after the specified number of characters have been sent to the terminal device. The range for <expr> is 1 to 132.

```
Example:  10  LINE 96
              (Sets the maximum line length of the system's
              terminal device to 96 characters).
```

5.1.8 MAT Statement

```
Format:  10  MAT <array>=[+-]<item>[+-<item>[...+-<item>]]]
```

This statement is used to set each element of <array> equal to a value determined by the specified equation. <Item> may be either another <array> or a numeric expression enclosed in parenthesis. All <arrays> must have the same number of elements, or a dimension mismatch error will occur. The same array may be referenced on both sides of the equal sign; however, the destination <array> is used as a work area; therefore, its values will change as the expression is evaluated. Thus, MAT A=B+A is not the same as MAT A=A+B. (MAT A=B+A would be the same as MAT A=B+B).

```
Examples:  10  MAT A=(1)
           20  MAT A=A+B-C+(5)
           30  MAT A=-A
```

5.1.9 MAT READ Statement

Format: 10 MAT READ <array 1>[,<array 2>[...,<array n>]]

This statement will read the specified arrays from DATA statements (see READ and DATA statements). The lowest element in the array is read first (see example).

```
Example:  10  DIM A(2,2),B(1)
          20  MAT READ A,B
          30  FOR I=0 TO 2 : FOR J=0 TO 2 : PRINT A(I,J);
              : NEXT J : PRINT : NEXT I : PRINT B(0);B(1)
          40  DATA 1,2,3,4,5,6,7,8,9,10,11
          RUN
           1 2 3
           4 5 6
           7 8 9
           10 11
```

5.1.10 PACK Statement

Format: 10 PACK (<mask>)<str variable> FROM <expr>

The PACK statement is used to compress numeric data into a string variable. The mask is used to indicate the maximum size of the result string and the number of decimal places in the result. The format of the mask is the same as CONVERT. The length of the result string is calculated by the following formula:

Number of bytes = INT((Number of significant digits indicated by <mask> + 1) / 2)

This two for one compression will require less memory for storage of numbers with a few significant digits and is particularly useful for size reduction of mass storage records (see also "UNPACK").

```
Example:  X=314.159
          PACK (###.##) A$ FROM X
          PRINT LEN(A$)
           3
```

5.1.11 READ and DATA Statements

Formats: 10 READ <variable 1>[,<variable 2>[, ... <variable n>]]
20 DATA <expr 1>[,<expr 2>[, ... <expr n>]]
(The variables and expressions may be either string or numeric.)

The READ and DATA statements are used to load predetermined data into program variables. The READ statement will assign the values contained in the DATA statement to the variables named in the READ statement. The first variable in the first READ statement will be assigned the first value contained in the first DATA statement in the program. Each succesive variable contained in a READ statement will be assigned the next value in a DATA statement. If all values in a DATA statement are used, the next DATA statement in the program is used. Care should be taken that the type of variable and the type of the value agree (i.e., a numeric value for a numeric variable and a string value for a string variable) or an error will occur.

Example:

```
10  READ X,A$,B,Z$
20  DATA 23,"TEST"
30  DATA 50+X,A$+"2"
40  PRINT X,A$,B,Z$
RUN
 23      TEST      73      TEST2
```

5.1.12 RESTORE Statement

Formats: 10 RESTORE [<statement number>]
10 RESTORE ERROR

In the first form, the RESTORE statement is used to instruct Business BASIC to begin using a particular DATA statement. If the optional statement number is given, the DATA statement indicated will be used by the next READ statement. If no statement number is given, the first DATA statement in the program will be used by the next READ statement.

The RESTORE ERROR statement will cause the effect of any existing ON ERROR statement to be cleared and error control to be returned to BASIC (see "ON ERROR").

Examples:

```
10  RESTORE
70  RESTORE 85
90  RESTORE ERROR
```

5.1.13 UNDIM Statement

Format: 10 UNDIM <variable 1>[,<variable 2> ... [,<variable n>]]
(<Variable> may be string or numeric array names.)

The UNDIM statement is used to free up the space occupied by strings and/or numeric arrays. The indicated strings and/or numeric arrays are deleted and all other variables are moved so that all variables will occupy a contiguous section of memory. All data contained in the variables named in the UNDIM will be lost. Note: UNDIM may not be used within either a FOR-NEXT loop or a user defined function, or a control stack error will be occur.

Example:

```
PRINT FREE(0)
 7817
DIM A(15),B$(12)
PRINT FREE(0)
 7703
UNDIM A,B$
PRINT FREE(0)
 7817
```

5.1.14 UNPACK Statement

Format: 10 UNPACK (<mask>) <variable> FROM <str variable>

The UNPACK statement is the inverse of the PACK statement. This statement is used to take a PACKed number in a string and turn it back into a numeric variable. The mask is used to indicate the format of the PACKed number. The format of the mask is the same as CONVERT.

Example:

```
X=314.159
PACK (###.##) A$ FROM X
UNPACK (###.##) X FROM A$
PRINT X
 314.16
```

5.2 Program Control Statements:

5.2.1 END Statement

Format: 10 END

This statement terminates execution of a program. There is no way to CONTinue execution after an END.

Example: 9999 END

5.2.2 EXIT Statement

Format: 10 EXIT <statement number>

This statement functions in the same way as a GOTO statement. In the process it terminates the currently active FOR-NEXT loop. It must be used to branch out of a FOR-NEXT loop. Using a GOTO to branch out of a FOR-NEXT loop will result in a control stack error. This is because a FOR-NEXT loop places information on the internal control stack and a GOTO will not clear this information. (In earlier versions of Maxi-BASIC, the EXIT cleared ALL active FOR-NEXT loops. This version clears only the currently innermost loop).

Example:

```
10  FOR I=1 TO 3
20    FOR J=1 TO 10
30      IF J=3 THEN EXIT 60
40      PRINT I,J
50      NEXT J
60    NEXT I
RUN
 1       1
 1       2
 1       3
 2       1
 2       2
 2       3
 3       1
 3       2
 3       3
```

5.2.3 FOR and NEXT Statements

```
Format:  10  FOR <variable>=<expr 1> TO <expr 2> [STEP <expr
              3>]
         50  NEXT [<variable>]
```

These statements are used to establish iterative loops. The FOR statement, during the first iteration, has no effect except to assign the value of <expr 1> to <variable>. Control is then passed to the statement following the FOR. When the NEXT statement is encountered, several things occur. First, the value of the optional <expr 3> (or one if no STEP is present) is added to <variable>. Then a comparison is made between <variable> and expr2. If <variable> is greater, control is passed to the statement following the NEXT statement; otherwise, control is passed to the statement following the FOR.

Example:

```
10  FOR I=1 TO 10 STEP 3
20    PRINT I
30    NEXT
```

(Note that <variable> is optional in the NEXT. If it is present, a check will be made for proper nesting).

```
RUN
 1
 4
 7
 10
```

FOR-NEXT loops may be multiply nested. Care must be taken so that FOR-NEXT loops are contained totally one within the other. No overlap of FOR-NEXT loops may occur.

5.2.4 GOSUB Statement

Format: 10 GOSUB <statement number>

The GOSUB statement is used to pass control to the <statement number> indicated and establish the linkage necessary to come back to the statement following the GOSUB. (Used with RETURN).

Note: The GOSUB statement is not a directly executable statement; that is, it may not be executed in the immediate mode.

Example: 10 GOSUB 1055

5.2.5 GOTO Statement

Format: 10 GOTO <statement number>

The GOTO statement passes control to the indicated statement number.

Example: 10 GOTO 1905

5.2.6 IF-THEN-ELSE Statement

Format: 10 IF <expr> THEN <statement> [ELSE <statement>]

This statement allows conditional testing and selective statement execution based on this conditional testing. The <expr> (which can be either a simple compare or a complex boolean condition using AND, OR and NOT) is evaluated and if it is true, the statement following the THEN is executed. If the condition is false and there is an ELSE clause, the statement following the ELSE will be executed. If no ELSE clause is present and the condition is false, the next numbered statement will be executed. The statements contained in the THEN or ELSE clauses may be any valid statement (including another IF). If the statement in the THEN or ELSE clause is a GOTO, the GOTO is optional (only a statement number is necessary).

Example:
```
A=0 : B=1
IF A=0 OR B=0 THEN PRINT "YES" ELSE PRINT "NO"
YES
IF A AND B THEN PRINT "YES" ELSE PRINT "NO"
NO
(Note: This is a pure boolean operation. See
"Boolean Operators").
IF A>2 THEN PRINT "YES" ELSE IF B=1 THEN PRINT
     "B IS ONE" ELSE PRINT "NO"
B IS ONE
A$="A"
IF A$<"B" AND B THEN PRINT "TRUE"
TRUE
```

5.2.7 ON ERROR Statement

Format:
```
10  ON ERROR (<str variable 1>,<str variable 2>)
      <statement>
```

The ON ERROR statement is used to allow the user's program to retain control when an error occurs during program execution. Previous versions of Maxi-BASIC would terminate execution upon detection of any error. If Business BASIC detects any type of error, a check will be made to see if an ON ERROR statement has been executed. If so, the <statement number> at which the error occured will be placed in <str variable 1>, the error message will be placed in <str variable 2>, and control will be passed to the <statement> contained in the ON ERROR. Control is always passed to the last ON ERROR statement executed. If no ON ERROR statement is executed and Business Basic detects an error, it will terminate execution of the program (see also "RESTORE ERROR").

Example:
```
10   DIM E$(11)
20   ON ERROR (L$,E$) GOTO 2000
30   PRINT "Enter code number: ";
40   ENTER 1000,X$,X
50   PRINT
60   C=VAL(X$)
70   RESTORE ERROR
80   PRINT LOG(-1)
90   STOP
2000  PRINT "Invalid code number."
2010  GOTO 30
```

```
2030  END
RUN
Enter code number: XYZ
Invalid code number.
Enter code number: 123
Out of bounds value error in line 80
```

5.2.8 ON-GOSUB Statement

Format: 10 ON <expr> GOSUB <statement number 1>[,<statement number 2>...[,<statement number n>]]

The ON-GOSUB statement is used to transfer control to one of several statements in a program, dependent upon the value of <expr>. The linkage is also established to allow subsequent transfer back to the statement following the ON-GOSUB (using a RETURN). The value of <expr> must be an integer greater than zero. If the value of <expr> is one, control will be passed to the first statement number named. If the value of <expr> is two, control will be passed to the second statement number. Likewise, if the value of <expr> is n, control will be passed to the nth statement number. If the value of <expr> is greater than n, an error will occur.

Note: The ON-GOSUB statement is not a directly executable statement; that is, it may not be executed in the immediate mode.

Example: 10 ON X GOSUB 30,40,50,60
(Control will be passed to statement 30 if X=1, statement 40 if X=2, statement 50 if X=3, or statement 60 if X=4).

5.2.9 ON-GOTO Statement

Format: 10 ON <expr> GOTO <statement number 1>[,<statement number 2>...[,<statement number n>]]

The ON-GOTO statement functions exactly the same as ON-GOSUB, except that ON-GOTO does not establish the linkage necessary for a RETURN; otherwise the two statements are equivalent. The value of <expr> is checked and control is transfered to the appropriate statement number.

Example: 10 ON X GOTO 30,40,50,60

5.2.10 RETURN Statement

Format: 10 RETURN [<variable>]

The RETURN statement is used to transfer control back to the statement following the last executed GOSUB or ON-GOSUB. It is also used to transfer control back to the program after execution of a user defined function. The optional <variable> is used only with the latter type, and indicates the value to be sent back to the program (see "User Defined Functions"). Control may be passed to the same subroutine (via a GOSUB or ON-GOSUB) from several places in a program. The RETURN statement will be able to correctly identify the point from which control was transfered, and return control to that point.

Example:
```
10  GOSUB 500
20  PRINT "LINE 20"
30  GOSUB 500
40  PRINT "LINE 40"
50  STOP
500  PRINT "LINE 500"
510  RETURN
RUN
LINE 500
LINE 20
LINE 500
LINE 40
STOP IN LINE 500
```

5.2.11 STOP Statement

Format: 10 STOP

The STOP statement terminates execution of a program. The message "STOP IN LINE xxx" is then displayed (if the STOP is not the last line of the program) where xxx is the line number of the statement following the STOP. The program may be continued after execution of a STOP by typing CONT.

Example: `1090 STOP`

5.2.12 TRACE Statement

Formats:
```
10  TRACE START [LINE]
10  TRACE STOP
```

The TRACE START statement is used to instruct Business BASIC to list each statement on the terminal device before it is executed. This statement facilitates easy debugging of the user's program. If the option LINE is specified, only line numbers are printed.

The TRACE STOP statement instructs Business BASIC to terminate listing of statements initiated by a TRACE START.

5.3 General Input/Output Statements:

5.3.1 CURSOR Statement

Format: 10 CURSOR <expr 1>[,<expr 2>]

The CURSOR statement is used to position the cursor of the TV-64 controller at a specified location. If only <expr 1> is present it may have a value of zero thru 1023. The TV-64 display is then treated as a single dimensioned array and the cursor placed at the indicated position. If <expr 1> and <expr 2> are present, <expr 1> may have a value of zero thru fifteen and <expr 2> may have a value of zero thru sixty-three. The TV-64 display is then treated as a two dimensional array and the cursor placed at the indicated position. The value of <expr 1> indicates the desired row and the value of <expr 2> indicates the column desired.

```
Example: 10  CURSOR 3,15
         20  CURSOR 521
```

5.3.2 ENTER Statement

Format: 10 ENTER <expr>,<str variable>,<variable>

The ENTER statement is used for timed input of data from the keyboard. <Expr> indicates the number of tenths of seconds to await input. If <expr> has a value of zero, then the time limit is set to infinity. The limits on the value of <expr> are zero to 65535. <Str variable> indicates the string variable in which to place the incoming data. <Variable> indicates the variable in which to place the number of tenths of seconds actually used in completing the input. ENTER will transfer control to the statement following the ENTER after one of three events occurs. First, control is transferred if the input is not completed within the allotted time. In this case, <variable> will be set to zero. Second, control is transferred if a carriage return is entered. In this case, <variable> will contain the time used. Finally, control will be transferred if the number of characters entered is equal to the dimensioned size of <str variable>. In this case, <variable> will contain the time used. In the final method, no carriage return is required. This is useful for controlling maximum input length. Note that ENTER, like INPUT1, will not generate a carriage return/line feed at the end of the user's input.

Example: 10 ENTER 50,A$,N

5.3.3 ENTER KEYIN Statement

Format: 10 ENTER KEYIN <expr 1>,<expr 2>;<str variable>

Formatted input is achieved by using the ENTER KEYIN statement. Formatted input is essentially a method of "screening" keyboard input as it is being typed. Unwanted characters are simply ignored. This may be used to input such things as dates, ID numbers, or dollar amounts.

<Expr 1> is the field type. This specifies what type of input is to be expected, and must be an integer from 0 to 6. Field types are:

- 0 = Any ASCII character string
- 1 = Date (format MM/DD/YY; slashes are automatic)
- 2 = Dollar amount (may be negative)
- 3 = Digits (accepts only 0-9)
- 4 = Any ASCII character except carriage return
- 5 = Digits (accepts 0-9 or carriage return)
- 6 = Dollar amount (must be positive)

<Expr 2> is the length of the field, where 1 <= <expr 2> <= LINE LENGTH. If <expr 2> > LINE LENGTH, <expr 2> defaults to LINE LENGTH. ENTER KEYIN will display a number of periods corresponding to this value when executed; this gives the user an indication of what the maximum field length is. When input has finished, any excess periods will be written over with spaces.

<Str variable> will contain the returned field. If <str variable> is a substring (e.g., A$(2,6)), it will be buffered with spaces if the input field is shorter than the substring.

If any control character (except carriage return) is typed as the first character of the field, that character will be stored is str variable, and control will be passed on to the next statement. If a carriage return is typed as the first character of the field, <str variable> will be returned null (length zero) and control will be passed on to the next statement.

Example:
```
10  ENTER KEYIN 1,8;A$
20  ENTER KEYIN 3,5;N$
30  ENTER KEYIN 4,2;S$
```

Line 10 would be used to enter a date into A$. Line 20 would be used to enter a five digit ID number into N$. Note that since carriage return is ignored by field type 3, the returned value could contain no less than the specified (in this case five) number of digits. Line 30 is similar, except that input is alphanumeric; in this case, S$ might contain a two letter state abbreviation (such as CA).

5.3.4 IMAGE Statement

Format: 10 IMAGE<mask 1>[<mask 2>[<mask 3>[... <mask n>]]]

The IMAGE statement provides the output format to be used for variables and constants in a PRINT USING statement. The first value in the PRINT USING uses <mask 1>, the second value uses <mask 2>, and so on with the nth value using <mask n>. The masks are made up of combinations of the following characters:

- # is a replacement character, i.e., in the resulting output a character of a variable will replace the #.
- . is used to indicate placement of the decimal point in the resulting output.
- $ is used to indicate that a dollar sign is to be placed to the immediate left of the field.
- \+ is used to indicate desired output of a sign, whether positive or negative, with a numeric variable.
- \- is used to indicate output of a minus sign for negative numeric variables and suppression of plus signs.
- \ is used to indicate output of a carriage return and line feed.
- ↑ is the replacement character for exponents in scientific notation.
- ; is used to separate masks. It has no effect on the resulting output.

String constants may also occur within the IMAGE statement. Spaces in the IMAGE are treated as literal spaces and need not be quoted.

Example:

```
10  X=15 : A$="Barrels"
20  PRINT USING 30;X,A$
30  IMAGE"Qty on hand is "####.## #########
RUN
Qty on hand is   15.00 Barrels
```

5.3.5 INPUT Statement

Format: 10 INPUT[1][<str constant>,] <variable 1>[,<variable 2>[, ... <variable n>]]

The INPUT statement is used to assign values obtained from the keyboard to the named variables. The variables may be string or numeric. If present, <str constant> will be printed on the terminal as a prompt. If no <str constant> is present, a "?" will be used. The INPUT1 statement is identical to the INPUT statement except that INPUT1 will suppress echoing of the carriage return/line feed at the end of the users input.

```
Example:  10  INPUT "Enter time, item: ",A,X$
          20  INPUT1 B,Z$
```

During execution of an INPUT statement, the user may provide multiple values on the same line by separating them by commas. Note, however, that the value assigned to a string variable is not terminated by a comma, but a carriage return. If the user gives fewer values than required to satisfy the INPUT statement, the prompt "??" will appear.

```
Example:  10  DIM A$(15)
          20  INPUT X,A$,Y
          30  PRINT X,A$,Y
          RUN
          ?23,test,45          (typed by user)
          ??93                 (typed by user)
           23     test,45        93
```

5.3.6 KEYIN Statement

Format: 10 KEYIN <str variable>

The KEYIN statement is used to input single key data entries. The value of any key pressed will be placed in the string variable named. This statment differs from INPUT in that data from any key (except ESC, unless break has been disabled using the BRK function) will be placed in the variable. INPUT does not allow entry of certain keys (for example carriage return).

```
Example:  10  KEYIN A$
```

5.3.7 OUT Statement

Format: 10 OUT <expr 1>,<expr 2>

The OUT statement is used to output a specific value to a particular port. <Expr 1> indicates the port number and <expr

2> indicates the value. <Expr 1> and <expr 2> may have a value of zero thru 255.

Example: 10 OUT 5,200

5.3.8 PRINT Statement

```
Formats:  10  PRINT [%<format string>,] <expr 1>[[,]<expr 2>
                [ ... [,]<expr n>]]
          10  #[%<format string>,] <expr 1>[[,]<expr 2>[[,]
                ... <expr n>]]
```

The PRINT statement is used to output values to the terminal. The # sign may be used in place of the word PRINT. The expressions may be string or numeric. If the expressions are separated by commas, the values will be printed in eight fields of eight characters. For expressions separated by a semicolon, a space preceeds a numeric expression; no space preceeds a string expression. If the expressions are separated by a backslash (\), a carriage return/line feed will be printed between the values. Note that the delimiters are optional, and that multiple consecutive delimiters may be used.

If a value will not fit on the current output line, it will be placed on the next line. Output of the values is in the default format, unless formating is specified. The default format is initially free format. A format string may appear anywhere in the PRINT statement and begins with a % character. A format string is made up of optional format characters followed by format specifications. The format characters are:

- C — places commas to the left of the decimal as needed
- $ — places a dollar sign to the left of the value
- Z — suppresses trailing zeroes
- ? — makes this format string the default format

Valid format specifications are:

- nFm — Floating point format. The value is printed in an n character field with m digits to the right of the decimal point.
- nI — Integer format. The value is printed in an n character field. (An error will occur if a non-integer value is used).

nEm Scientific format. The value is printed in an n character field with m digits to the right of the decimal point in scientific notation (i.e., a mantissa and exponent).

(Note: In each format specification, the n character field must include any commas and/or any dollar signs).

Values will be rounded if necessary to fit the format specification.

Example:
```
X=1234.5609
Y=5678.9035
PRINT X,Y
 1234.5609       5678.9035
#X\\Y
 1234.5609

 5678.9035
PRINT X;Y
 1234.5609 5678.9035
PRINT %7F2;X
1234.56
PRINT %C8F2;X
1,234.56
PRINT %10E0;X
123456E-02
X=1234
PRINT %C$6I;X
$1,234
```

5.3.9 PRINT HEX Statement

Format: 10 PRINT HEX <str expr 1>[,<str expr 2>[, ... <str expr n>]]

This PRINT statement is used to output the value of the string expressions in hexadecimal format. Each character of the string expressions is converted to its corresponding two character hexadecimal code and sent to the terminal device.

Example:
```
A$="10AZ"
PRINT HEX A$
3130415A
```

HEX may be intermixed with other data to be printed (e.g., when examining packed data fields).

Example:
```
X=314.159
PACK (###.##) B$ FROM X
PRINT "PACKED DATA: ";HEX B$
```

```
PACKED DATA:  314150
```

5.3.10 PRINT USING Statement

```
Formats:  10  PRINT USING <statement number>;<expr 1>[,<expr 2>
                [, ... expr n]][;]
          10  PRINT USING <str expr>;<expr 1>[,<expr 2>[, ...
                <expr n>]][;]
```

The PRINT USING statement is used to provide formatted output of data. The statement provides a list of expressions (numeric or string) whose value is to be PRINTed using the masks provided by the IMAGE statement referred to by the statement number (see IMAGE statement). The masks may optionally be contained in <str expr>. A semicolon may be used as the last character of a PRINT USING statement to suppress the carriage return/line feed at the end of the line.

```
Example:  10  DIM B$(50)
          20  X=15 : A$="barrels"
          30  B$="############## ####.##  #########"
          40  PRINT USING B$;"QTY ON HAND IS",X,A$
          QTY ON HAND IS  15.00  barrels
```

5.4 File Handling Statements

5.4.1 CLOSE Statement

```
Format:  10  CLOSE (<expr>,<variable>)
```

The CLOSE statement is used to disassociate a logical file number from a physical file, thus allowing the logical file number to be assigned to another physical file. <Expr> indicates the logical file number to be CLOSEd. If the file did not exist at OPEN time, an entry will be placed in the directory of the tape on which the physical file is located. After execution of the CLOSE, <variable> will contain a status code (see "Appendix A").

```
Example:  CLOSE (4,E)
```

(Note: Files are NOT automatically CLOSEd upon termination of the program. This allows one program to pass an open file to another).

5.4.2 GET Statement

Format: 10 GET (<expr 1>,<variable>,<str variable>,<expr 2>
[,<expr 3>])

The GET statement is used to retrieve a record from the specified logical file. <Expr 1> indicates the logical file number. <Expr 2> indicates the record number within the file to be retrieved. Note that the record number is not the hardware block id number used by PHIMON, but rather a logical record number. Record numbers are specified relative to the beginning of the named file. This frees the user from concern of hardware block id's. The first logical record number in a file is record number zero. The contents of the specified record are placed in <str variable>. The length of <str variable> is set to the smallest of the following values: the dimensioned size of <str variable>, or the value of optional <expr 3>. If the length of <str variable> is greater than 256 bytes, as many records will be read as are needed to fill the string. A status code is placed in <variable> after execution of the GET.

Example:

```
10  DIM A$(256),F$(8)
20  F$="TEST.DA"
30  OPEN (0,E,F$,3,1,1)
40  IF E THEN STOP
50  A$="THIS IS A DATA RECORD"
60  PUT (0,E,A$,1)
70  IF E THEN STOP
80  GET (0,E,A$,1)
90  IF E THEN STOP
100 PRINT A$
110 GET (0,E,A$,1,7)
120 IF E THEN STOP
130 PRINT A$
140 UNDIM A$
150 DIM A$(11)
160 GET (0,E,A$,1)
170 IF E THEN STOP
180 PRINT A$
RUN
THIS IS A DATA RECORD
THIS IS
THIS IS A D
```

5.4.3 OPEN Statement

Format: 10 OPEN (<expr 1>,<variable>,<str variable>,<expr 2>,
<expr 3>[,<expr 4>])

The OPEN statement is used to associate a physical file with a logical file number. <Expr 1> indicates the logical file number to be used. The valid logical file numbers are zero thru nine. Up to ten files may be in use at one time; however, only one output or input/output file may be in use on a given physical drive at one time. <Str variable> contains the name of the file including the PHIMON extension (see pages 11 and 47 of the PHIMON manual). <Expr 3> indicates the physical drive in which the file will be used. Valid drive numbers are zero thru seven. The type of file is indicated by <expr 2>. The types of files are:

- Type 1: Output file. Records may only be PUT to this file. It must not already exist on the media mounted in the indicated drive.
- Type 2: Input file. Only GET's may be issued to this file. It must already exist on the media mounted in the indicated drive.
- Type 3: Input/output file. Both GET's and PUT's may be issued to this file. No assumption is made as to whether or not the file exists on the media mounted in the indicated drive.

The optional <expr 4> indicates the number of records in the file. For input-type files it is ignored. For output-type files, a check is made to determine if sufficient space exists to establish the file. If not, a particular status code is placed in the named variable. For input/output files that already exist at the time the OPEN is executed, no action is taken. For input/output files that do not already exist, a check is made to verify that the tape has sufficient space. If no file size is specified for output files or input/output files that do not already exist, a default value equal to the maximum available free space on the tape is assumed. (Note that for files OPENed in this manner the file size will be adjusted when the file is CLOSEd to reflect the actual number of records used). A status code is placed in <variable> after execution of an OPEN statement.

Example:

```
DIM F$(8)
LET F$="TEST.DA"
OPEN (0,E,F$,1,2)
```

5.4.4 PURGE Statement

Format: 10 PURGE (<expr>,<variable>)

The PURGE statement is used to CLOSE a file and delete the file's name from the directory on its associated drive. The logical file number, indicated by <expr>, is disassociated from the physical file, just as in a CLOSE. A PHIMON/DISKMON delete is then is used to remove the file name entry from the directory. A status code is then placed in the named variable.

Example: PURGE (0,E)

5.4.5 PUT Statement

Format: 10 PUT (<expr 1>,<variable>,<str variable>,<expr 2> [,<expr3>])

The PUT statement is used to place a record on the specified file. <Expr 1> indicates the logical file number. The contents of <str variable> are placed into the record number indicated by <expr 2>. Optionally, <expr 3> indicates the number of characters of <str variable> that are to be placed in the file. For tape files, records must initially be put into a file sequentially. As an example, record number five must already exist in the file before a PUT to record six can take place. (Note: the authors have been using random GET's and PUT's to a file after it has initially had blank records written into it. Further, we have also been using an update-in-place scheme, i.e., multiple PUT's to the same record, with no detrimental results. The user is cautioned that in tape files, write errors might cause subsequent records to be lost. It is therefore advised that suitable backup procedures be instituted for files used in this way).

Example:

```
10  DIM A$(256),F$(8)
20  F$="TEST.DA"
30  OPEN (0,E,F$,3,1,2)
40  IF E THEN STOP
50  A$="THIS IS A DATA RECORD"
60  PUT (0,E,A$,1)
70  IF E THEN STOP
80  PUT (0,E,A$,2,7)
90  IF E THEN STOP
100  GET (0,E,A$,1)
110  IF E THEN STOP
120  PRINT A$
130  GET (0,E,A$,2)
140  IF E THEN STOP
150  PRINT A$
RUN
```

THIS IS A DATA RECORD
THIS IS

5.4.6 REWIND Statement

Format: 10 REWIND (<expr>,<variable>)

The REWIND statement is used to physically position a tape at the begining of the logical file specified by <expr>. A status code is then placed in <variable>.

Example: REWIND (0,E)

5.5 Miscellaneous Statements

5.5.1 REM Statement

```
Formats: 10  REM[any comment]
         10  '[any comment]
```

This statement is used to place remarks in a program. It is ignored during execution of the program, but will appear in any LIST of the program.

Example: 10 REM THIS PROGRAM WRITTEN OCT. 15

5.5.2 SLEEP Statement

Format: 10 SLEEP <expr>

Causes the system to suspend processing for the number of seconds indicated by <expr>. <Expr> must fall into the range of 0 to 655.35 seconds. Precision is kept up to .01 second; thus, if <expr> had a value of 1.253, BASIC would wait for 1.25 seconds before continuing.

Example: SLEEP 1.25

CHAPTER 6

FUNCTIONS

The purpose of functions in Business BASIC is to allow the programmer to use implicit subroutine access within most statements. In other words, most references to a variable may be replaced with a reference to a subroutine and the value returned by that subroutine is used in the execution of the statement. Two types of functions are available in Business BASIC: system functions and user defined functions.

6.1 System Functions:

The system functions are used by replacing the reference to a variable in most statements with the name of the desired function followed by a parameter enclosed in parentheses. Business BASIC will execute the desired function and the result returned by the function will be used in the statement.

Example:
```
X=24.56
PRINT X,INT(X)
 24.56     24
Y=INT(X)+5
PRINT Y
 29
```

The system functions available in Business BASIC are as follows.

6.1.1 ABS Function

Format: ABS(<expr>)

The ABS function is used to obtain the absolute value of <expr>.

Example:
```
X=-354
PRINT ABS(X)
 354
```

6.1.2 ASC Function

Format: ASC(<str expr>)

The ASC function is used to obtain the numeric value of the first character of the indicated string.

```
Example:  A$="3"
          PRINT ASC(A$)
           51
          PRINT ASC("m")
           109
```

6.1.3 ATN Function

Format: ATN(<expr>)

The ATN function is used to obtain the arctangent of <expr>. The result is given in radians.

```
Example:  PRINT ATN(1)
           .78539818
```

6.1.4 BRK Function

Format: BRK(<expr>)

This function returns the current BREAK status. If CTL C and ESC are enabled, the return value will be one; otherwise, the return value will be 0. Depending on the value of <expr>, this function may also set BREAK status. If <expr> is less than zero, status is unchanged. If <expr> is zero, status will be set to zero and CTL C and ESC will no longer function. If <expr> is greater than zero, status will be set to one and CTL C and ESC will be enabled.

```
Example:  PRINT BRK(0)
           0
          (Would also disable CTL C and ESC).
```

6.1.5 CALL Function

Format: CALL(<expr 1>[,<expr 2>])

The CALL function is used to pass control to a machine language subroutine. <Expr 1> indicates the address of the machine language subroutine to be CALLed. The value of optional <expr 2> is converted to an integer and placed in the DE register pair before the machine language routine is executed. The machine language routine should place a value, to be returned to the main program, in the HL register pair. This value is then returned to the statement which referenced the CALL function.

Example: PRINT CALL(2051,300)
600
(This would place the hex value 012C in the DE register pair and then execute a fictitious machine language routine at hex address 0803. The fictitious routine shifted the number in the DE register pair left one bit and placed the result, hex 0258, in the HL register pair. This value was then returned to the PRINT statement).

6.1.6 CHR$ Function

Format: CHR$(<expr 1>[,<expr 2>])

This function is functionally the inverse of the ASC function. It obtains the ASCII character represented by the value of <expr 1>. <Expr 2> specifies the number of times the character is to be returned. If <expr 2> is not specified, the default value is 1.

Example: PRINT CHR$(65);" ";CHR$(66,5)
A BBBBB

6.1.7 COS Function

Format: COS(<expr>)

The COS function is used to obtain the cosine of <expr>. <Expr> must be expressed in radians.

Example: PRINT COS(.234)
.9727467

6.1.8 DEG Function

Format: DEG(<expr>)

The DEG function is used to convert radians to degrees. <Expr> is expressed in radians and the output value is given in degrees.

Example: PRINT DEG(3.1415926)
179.99999

6.1.9 DIR Function

Format: DIR(<expr>)

The DIR function is a string function which returns as its value the directory of the specified drive. <Expr> must be an integer from 0 to 7.

Example:

```
10  DIM A$(1024)
20  A$=DIR(0)
30  E=LEN(A$)/16
40  #DIR(0)+CHR$(14);"No. of entries =";E
RUN
TEST   .BA     1   BASIC .GO    79
Number of entries = 2
```

6.1.10 EXAM Function

Format: EXAM(<expr>)

The EXAM function is used to obtain the value of the contents of a particular memory location. <Expr> indicates the address of the desired location.

Example:

```
PRINT EXAM(2)
 224
```

6.1.11 EXP Function

Format: EXP(<expr>)

The EXP function is used to obtain the value of e raised to a specified power. <Expr> indicates the desired power to which e is to be raised.

Example:

```
PRINT EXP(3)
 20.085535
```

6.1.12 EXP10 Function

Format: EXP10(<expr>)

The EXP10 function is used to obtain the value of 10 raised to a specified power. <Expr> indicates the power to which 10 is to be raised.

Example:

```
PRINT EXP10(2)
 99.999999
```

6.1.13 FREE Function

Format: FREE(0)

The FREE function is used to obtain the number of unused bytes remaining in memory. A dummy parameter of zero is always used.

```
Example:  PRINT FREE(0)
           7817
```

6.1.14 INP Function

Format: INP(<expr>)

The INP function is used to obtain the data available from the specified input port. <Expr> indicates a port number to which a hardware input instruction is to be issued.

```
Example:  PRINT INP(0)
           13
```

6.1.15 INT Function

Format: INT(<expr>)

The INT function is used to obtain the greatest integer value of <expr>.

```
Example:  X=23.45
          PRINT INT(X)
           23
          X=-12.34567
          PRINT INT(X)
           -13
```

6.1.16 LEN Function

Format: LEN(<str expr>)

The LEN function is used to obtain the length of <str expr>. The value returned is a count of the actual number of characters currently in <str expr>.

```
Example:  DIM A$(15)
          A$="TEST"
          PRINT LEN(A$)
```

```
 4
A$=A$+"ING"
PRINT LEN(A$)
 7
```

6.1.17 LOG Function

Format: LOG(<expr>)

The LOG function is used to obtain the natural logarithm of <expr>.

Example:
```
PRINT LOG(23)
 3.1354941
```

6.1.18 LOG10 Function

Format: LOG10(<expr>)

The LOG10 function is used to obtain the base ten logarithm of <expr>.

Example:
```
PRINT LOG10(100)
 2
```

6.1.19 PI Function

Format: PI

Returns the value of Pi to eight digits.

Example:
```
PRINT PI
 3.1415926
```

6.1.20 POS Function

Format: POS(<expr>)

Used to obtain the current position of the screen cursor. If <expr> is less than zero, then the absolute position (0 to 1023) will be returned. If <expr> is zero, the row position (0 to 15) will be returned. If <expr> is greater than zero, the column position (0 to 63) will be returned.

Example:
```
CURSOR 1,3 : X=POS(-1) : Y=POS(0) : Z=POS(1)
PRINT X;Y;Z
 66 1 3
```

6.1.21 RAD Function

Format: RAD(<expr>)

The RAD function is used to convert degrees to radians. <Expr> is expressed in degreess and the output value is given in radians.

```
     Example:  PRINT RAD(180)
                3.1415927
```

6.1.22 RND Function

Format: RND(0)

The RND function is used to obtain a psuedo randam number with a value between zero and .99999999. The dummy parameter zero is always used.

```
     Example:  PRINT RND(0)
                .01234912
```

6.1.23 SCH Function

Format: SCH(<str variable>,<str expr>)

This function will search <str variable> for <str expr>. If found, the return value will be the position of the first character where <str expr> exists within <str variable>; otherwise, the return value will be 0. Note that <str variable> may be subscripted.

```
     Example:  A$="TEST"
               PRINT SCH(A$,"T");SCH(A$,"ES");SCH(A$,"Z");
                     SCH(A$(2),"T");SCH(A$(2,3),"T")
                1 2 0 4 0
```

6.1.24 SGN Function

Format: SGN(<expr>)

The SGN function is used to obtain an indication of the sign of <expr>. If the value of <expr> is positive, a value of one is returned. If the value of <expr> is zero, a value of zero is returned. If the value of <expr> is negative, a value of minus one is returned.

```
Example:  X=10 : Y=0 : Z=-15
          PRINT SGN(X),SGN(Y),SGN(Z)
           1     0     -1
```

6.1.25 SIN Function

Format: SIN(<expr>)

The SIN function is used to obtain the sine of <expr>. <Expr> must be expressed in radians.

```
Example:  PRINT SIN(23)
          -.8462207
```

6.1.26 SQRT Function

Formats: SQRT(<expr>)
SQR(<expr>)

The SQRT function is used to obtain the positive square root of <expr>.

```
Example:  X=81
          PRINT SQRT(X)
           9
```

6.1.27 STR$ Function

Format: STR$(<expr>)

The STR$ function is used to obtain a string containing the character representation of <expr>.

```
Example:  X=23.50
          A$=STR$(X)
          A$=A$+" each"
          PRINT A$
           23.50 each
```

6.1.28 TAB Function

Format: TAB(<expr>)

The TAB function is used in PRINT statements to specify that output is to begin at a particular column (as specified by <expr>).

```
Example:  PRINT "TEST";TAB(10);"*"
          TEST      *
```

6.1.29 TAN Function

Format: TAN(<expr>)

The TAN function is used to obtain the Tangent of <expr>. <Expr> must be expressed in radians.

```
Example:  PRINT TAN(.78539818)
           1.0000001
```

6.1.30 VAL Function

Format: VAL(<str expr>)

The VAL function is functionally the inverse of the STR$ function. It is used to obtain the numeric value of the characters of <str expr>.

```
Example:  A$="23.7865"
          X=VAL(A$)+20
          PRINT X
           43.7865
```

6.2 User Defined Functions

There are two types of user defined functions: single statement and multiple statement functions. Usage of both types is the same as the usage of system functions. Definition of functions is acomplished by use of the DEF statement.

6.2.1 DEF Statement

```
Formats:  10  DEF FN<variable>(<variable 1>[,<variable 2>[, ...
                <variable n>]])
          10  DEF FN<variable>(<variable 1>[,<variable 2>[, ...
                <variable n>]])=<expr>
```

The named varaible may be a numeric or string variable. The first format is for multiple statement functions; the second is for single statement functions. The execution of a single statement function call evaluates <expr> and returns the value of that <expr>. The execution of a multiple statement function call executes the statements contained in the function definition and returns the value specified in the RETURN statement. The variables used as parameters are "local" to the function definition; that is, any change in the value of these variables within the function definition are not reflected within the main program (see example below, paying particular attention to the variable Y).

Example:
```
10  Y=45
20  PRINT FNA(Y,5),FNB(Y)
30  PRINT Y
40  STOP
50  DEF FNA(Y,X)
60  Y=Y+3
70  Z=Y*5
80  RETURN X
90  FNEND
100  DEF FNB(Z)=Z*20
RUN
 240      900
 45
```

6.2.2 FNEND Statement

Format: 10 FNEND

The FNEND statement is used to indicate the end of a multiple statement function definition (see "DEF").

CHAPTER 7

ADVANCED PRINT USAGE

7.1 Control Characters

Business BASIC has incorporated certain control characters to provide special output functions. Any time these control characters are encountered in a PRINT statement, the associated function will be performed. These characters are:

- Control H - ASCII Backspace - Move screen cursor left
- Control I - ASCII Horizontal Tab - Move screen cursor right
- Control K - ASCII Vertical Tab - Home screen cursor
- Control L - ASCII Form Feed - Clear screen
- Control M - ASCII shift out - Carriage return & Line feed
- Control Q - ASCII Device control 1 - Move screen cursor down
- Control R - ASCII Device control 2 - Move screen cursor up
- Control S - ASCII Device control 3 - Write the screen buffer to the TVC-64
- Control T - ASCII Device control 4 - Reverse scroll, clear top line and home screen cursor

The functions of these characters are suppressed during a LIST. A control M will generate a carriage return and line feed during execution and LIST.

7.2 Installing Printer Drivers

The printer driver supplied with Business BASIC starts at 0600 and use memory from 0600-08FF. The character to be printed should be passed in the Accumulator. The printer driver should not alter any registers. To install a different driver, simply replace the existing driver.

7.3 Output Device Selection

Business BASIC has the capability of selecting the device or devices to which output from PRINT statements is to be sent. The devices currently implemented are the TV-64 display and the printer. To cause output to be sent to the TV-64 display an OPEN(CRT,<variable>) statement is executed. To cause output to be sent to the printer an OPEN(PRINTER,<variable>) statement is executed. To suspend output to the TV-64 display a CLOSE(CRT,<variable>) is executed. Likewise, to suspend output to the printer a CLOSE(PRINTER,<variable>) is executed. When a program is executed, output is initially assumed to be sent to the TV-64 display. The variable specified in the OPEN or CLOSE for CRT or PRINTER currently has no function; it is implemented for future expansion.

APPENDIX A

FILE I/O ERROR CODES

The following are the status codes returned by the file statements:

Code	Error
0	No errors.
1	File not found. File does not exist on this media.
2	Not enough free blocks exist on this tape.
3	Duplicate file exists on this tape.
4	File has not been opened.
5	End of file or logical record ID is greater than file size.
7	File number already OPENed.
8	Invalid file number.
9	System error.
10	I/O mismatch (GET on output type file or PUT on input type file).
11	Invalid file type.
12	Too many output files on one drive.
14	An output type file was closed but no records were written to it. The file is closed but no directory entry is created.

The following codes indicate PHIMON errors (see Appendix C of the PHIMON manuel):

257	Equivalent to error 1 - CRC error.
258	Equivalent to error 2 - Block not found.
259	Equivalent to error 3 - Tape end or Jam.

The following codes indicate DISKMON errors (see Appendix A of the DISKMON manuel):

257	CRC error.
258	ID Read error.
259	Device error.

APPENDIX B

PROGRAM INSTALLATION (PHIMON VERSION)

The Business BASIC program is distributed on one cassette tape, which contains the Business BASIC program in audio form (1100 baud - Suding format). This form can be read using the PHIMON REad command. An audio cassette recorder is required to read this tape.

An operational PHIMON system is required to prepare Business BASIC for use. The following procedure is used to install the program, and assumes familiarity with use of the PHIMON system.

1. Start up the PHIMON system, using the PHIMON ROM ("PZB").
2. Apply the recommended patches to PHIMON described in "Appendix D".
3. Prepare the audio cassette recorder with the distribution tape.
4. Enter the command "RE 1000" and read in the audio file.
5. Save the program on the PHIMON system tape or on another tape used for program storage by entering the command "SA#n BASIC 1-117*5000", where n specifies the appropriate tape drive number.
6. Execute Business BASIC and verify the above steps by entering the command "RU#n BASIC", where n specifies the drive number containing BASIC.

Note: When RUnning Business BASIC, automatic execution of a source program may be effected by specifying a start address of 001270 (rather than 005000) in the SAve command. The program to be executed must be saved along with BASIC by using a PHIMON "SAve" command, not a BASIC "SAVE" command.

APPENDIX C

CONVERSION OF MAXI-BASIC SOURCE PROGRAMS

The following procedure can be used to convert Maxi-BASIC source programs to Business BASIC. It is assumed that Maxi-BASIC has been previously installed and is operating under PHIMON (refer to "Getting Started with PHIMON" in the PHIMON documentation for more information).

1. LOad Maxi-BASIC, then the source program to be converted.
2. STart execution of Maxi-BASIC (required to set internal pointers).
3. WRite to an audio cassette, starting at 063253 through the end of the source program. The end point can be determined by examining 062140 for location and 062141 for page.
4. LOad Business BASIC.
5. REad from the audio cassette generated in step 3, starting at the location indicated by the address stored in (020006)+1 byte.
6. STart execution of Business BASIC.
7. Using the SAVE command in Business BASIC language (not PHIMON), save the source program.

APPENDIX D

PHIMON CONSIDERATIONS

It is suggested that the following patches be applied to the PHIMON system being used with Business BASIC. Refer to the PHIMON documentation for detailed information on applying patches using the DTO command.

Location	Data	Explanation
345354	270	Compensates for a tape
	346	overrun problem which
346270	076037	may cause erroneous 258
	315150345	error codes.
	311	
340034	315361343	Fixes a potential
343361	076200	printer problem which
	323003	can develop if RESET
	257	is pressed while a
	323003	line is being printed.
	041104	
	341	
	311	
340364	373004	Fixes a problem with
340371	372000	use of the RUB key.

APPENDIX E

CONVERSION OF .0 AND .1 BUSINESS BASIC PROGRAMS

The only major source code difference between previous versions of Business BASIC and this version is the fact that all numeric constants are now stored in BCD, rather than ASCII, format. However, this does not make programs written under previous versions of BASIC incompatable. The LOAD, RUN (from media), and APP routines do the conversion automatically upon loading a program into memory. For large programs, the conversion may take several seconds. In order to eliminate this delay, programs need only be SAVEd after they have been converted. The next time the program is loaded, there will be no delay.

Note: Converted programs will invariably use more memory. This is because every numeric constant requires 6 bytes of storage space in the source code. It is suggested that applications where memory is critical be run under the old BASIC.

APPENDIX F

RESERVED WORDS

The following is a list of keywords which are converted to a single-byte code before being stored in program source code. When a LIST is produced, the codes are then converted back to the proper keywords. None of these words may be used as or embedded in a variable name. Each of the keywords below is preceded by the single-byte code which represents it.

92	#	87	DATA	CC	LEN	B4	REWIND
BA	'	FB	DATE	80	LET	CE	RND
E0	(	91	DEF	A7	LINE	C1	RUN
E2	*	D3	DEG	A1	LIST	C3	SAVE
E3	+	A0	DEL	C2	LOAD	DF	SCH
E5	-	BE	DELAY	DD	LOG	A3	SCR
E7	/	8B	DIM	D6	LOG10	CA	SGN
F4	<	F9	DIR	AB	MAT	CB	SIN
F0	<=	9B	ELSE	E6	MAX	AA	SLEEP
F1	<>	8D	END	E4	MIN	C4	SQR
F5	=	BF	ENTER	EE	MOD	C4	SQRT
F3	=<	FE	ERROR	A9	NAME	FA	START
F2	=>	DA	EXAM	83	NEXT	9F	STEP
F6	>	96	EXIT	F7	NOT	8C	STOP
EF	>=	DE	EXP	93	ON	AE	STR$
DB	ABS	D7	EXP10	B2	OPEN	97	STR$
EC	AND	95	FILL	ED	OR	9C	TAB
A8	APP	90	FN	94	OUT	C9	TAN
99	ASC	81	FOR	BC	PACK	9D	THEN
CF	ATN	D8	FREE	D4	PI	9E	TO
A2	AUTO	FC	FROM	D1	POS	C0	TRACE
D5	BRK	B6	GET	82	PRINT	B0	UNDIM
CD	CALL	89	GOSUB	C8	PRINTER	BD	UNPACK
9A	CHR$	88	GOTO	B5	PURGE	FD	USING
A4	CLEAR	C5	HEX	B7	PUT	98	VAL
B3	CLOSE	84	IF	D2	RAD	A5	XREF
A6	CONT	B9	IMAGE	85	READ	E0	[
BB	CONVERT	D9	INP	8F	REM	E1	↑
DC	COS	86	INPUT	AC	REM		
C7	CRT	C6	INT	8E	RESTORE		
B1	CURSOR	B8	KEYIN	8A	RETURN		

APPENDIX G

RUNNING BUSINESS BASIC ON A NON-2.5 MHZ SYSTEM

In order for Business BASIC to be run on a system with a clock speed other than 2.5 Mhz, the timing loops for the ENTER and SLEEP statements must be changed. To make the change, follow these steps.

1. RUn BBASIC from PHIMON/DISKMON
2. Type X=1515.2*s : Y=INT(X/256) : FILL 20607,X-Y*256,Y (Where "s" is the clock speed in Mhz [e.g., "4" for a 4 Mhz system]. This will cause the cursor blink speed to change.)
3. Type X=133.2*s : Y=INT(X/256) : FILL 22357,X-Y*256,Y (Where "s" is the clock speed in Mhz.)
4. Type CURSOR 2,0 : A=CALL(0) (This will cause an exit to PHIMON/DISKMON.)
5. SAve BBASIC using PHIMON/DISKMON. BBASIC will now have the correct timing constants.

APPENDIX H

BUSINESS BASIC ERROR MESSAGES

The following is a list of the error messages which can occur in Business BASIC, along with possible causes.

Syntax error - unrecognizable command; array accessed with wrong number of dimensions.

Memory full error - DIMension too large; too many variables; program too large; too many nested GOSUB's; too many nested FOR-NEXT loops.

No space error - Not enough contiguous free space exists on disk or tape to store program.

Bad arg error - Illegal characters after statement end; missing or illegal statement argument.

Dimension error - Attempt made to DIMension an array or string which already exists.

Function def error - Reference to undefined function.

Out of bounds value error - Parameter, subscript, or argument not within allowable bounds.

Type error - String expression or variable used when numeric required, or vice versa.

Format error - Value can not be PRINTed in field specified with "%" clause.

Line number error - Non-existant line number referenced.

Divide zero error - Division by zero attempted.

Continue error - CONT typed at some point other than after STOP or CTRL C, or after variable table has been cleared (via CLEAR, SCR, re-entering BASIC, or editing program statements).

No program error - RUN typed but no program exists; LOAD#, RUN#, or APP# attempted on program whic does not exist on specified drive.

Double def error - Attempt made to DEFine a function which has been DEFined earlier in the program.

Read error - Not enough DATA; DATA item of wrong type.

Control stack error - FOR without matching NEXT; NEXT without matching FOR; RETURN with no prior GOSUB; UNDIM within FOR-NEXT loop or user defined function.

Input error - Not enough items input; input items of wrong type.

Arg mismatch error - Incorrect number of arguments in a user defined function.

Missing quote error - No closing quote in a string literal.

Numeric ov error - Math operation resulted in a value greater than 9.9999999E+62 or less than -9.9999999E62.

Image error - Line number specified in PRINT USING is not an IMAGE statement.

Format overflow error - Value in PRINT USING statement will not fit in specifed IMAGE mask.

Dimension mismatch error - MAT arithmetic operation attempted on two or more arrays having a different total number of elements.

Internal stack ov error - Too many parenthesis levels; too many nested user defined function calls.

Length error - Too many numeric constants on one line (particularly in DATA statements).

No program name error - Attempt to use implied name feature when LEN(NAME)=0.

Sequence number overflow/overlap error - RENumber would result in line numbers greater than 65535 or which overlap; APPend attempted on a program which has a first line number less than or equal to the last line of the current program.

Variable name error - Variable name contains more than 32 characters.