Marinchip 9900 BASIC
Marinchip 9900 Extended Commercial BASIC
Marinchip 9900 Transaction BASIC

User Guide

by John Walker

Revised April 1979

Marinchip Systems 16 St. Jude Road Mill Valley, CA 94941 ■ (415) 383-1545

Table of contents

Table of contents

Table of contents

1. Introduction

Marinchip BASIC is an implementation of the BASIC language for the Marinchip 9900 computer system. All of the standard BASIC features are provided, and numerous extensions in the areas of string processing, file handling, and machine interface are implemented.

BASIC is intended to be used in an interactive program development mode, and is suited to the construction of programs that interact directly with the user at a terminal. Marinchip BASIC is equally well suited to beginning programmers learning their first programming language and experienced programmers developing application packages.

Marinchip BASIC may also be used as a powerful desk calculator.

1.1. Versions of Marinchip BASIC

Marinchip BASIC comes in three separate versions. The first is referred to simply as BASIC, and is a simple BASIC language. Real numbers in this version have between 6 and 7 digits of accuracy, and formatted output (PRINT USING) is not provided. Extended Commercial BASIC contains all of the features of the simple BASIC, but maintains 16 digits of accuracy in all calculations, provides PRINT USING, and contains a random-access file facility. Extended Commercial BASIC is most often used when extended precision and formatted output are required (typically for business applications). Transaction BASIC is an extended version of Extended Commercial BASIC which adds facilities which enable implementation of multi-terminal applications. Transaction BASIC is currently under development, so information relating to it in this manual is subject to change at any time.

This manual will identify those features present only in certain versions of BASIC. The codes for such items are:

B	Simple BASIC
EB	Extended Commercial BASIC
TB	Transaction BASIC

1.2. Structure of this manual

This manual will first discuss the elements that make up a BASIC program, then will explain each statement in detail. Finally, how to use BASIC will be explained. The facilities offered by BASIC for program entry, editing, and saving will be discussed.

1.3. Conventions

All examples given in this manual, and all references to BASIC keywords will appear in UPPER CASE TYPE. The BASIC system itself is insensitive to the case of data entered (except for quoted strings), so a program may be written without concern for the case of letters. In the descriptions of various components of the BASIC language, examples will be given with items enclosed in corner brackets, like <this>. Such an item indicates that in its place the user supplies an item as indicated by the contents of the corner brackets. For example, the line:

```
ZORCH <number>
```

indicates the user should write something that looks like:

```
ZORCH 18
```

An item enclosed in square brackets like [this] indicates that the item is optional. An item followed by an ellipsis (...), indicates that the item may be repeated any number of times. Hence, the line:

```
ZORCH <number>[,<number>,...]
```

describes things that look like:

```
ZORCH 23
ZORCH 118, 93, 23
ZORCH 9, 10
```

2. BASIC Language Components

2.1. Program

A PROGRAM is the unit which is executed in BASIC. A program consists of one or more LINES. The program being executed is stored in the BASIC "work area", and may be interactively modified by the user.

2.2. Line

A LINE is a part of a program which occupies one source line. Each line must begin with line number which must be between 0 and 32767 inclusive. Following the line number are one or more BASIC STATEMENTs. If more than one statement appears on a line, the statements must be separated by colons (:).

2.3. Statement

A STATEMENT is a declaration or command written according to the rules of the BASIC language. A statement may perform some action, or may declare data to be used in a program. Each program LINE must contain at least one STATEMENT. STATEMENTs may also be typed in without line numbers, in which case they will be executed immediately. Blanks are completely insignificant in BASIC, except when they appear within quotation marks. The user is encouraged, though, to insert blanks as an aid to readability. Chapter 3 of this manual discusses each BASIC statement in detail.

2.4. Constants

Constants represent data with a fixed value. A constant may either be numeric (for example 18 or 25.6) or a string (for example "Kaboom").

2.4.1. Numeric constants

Numeric constants may be written in several forms:

As an integer:

```
167 -239 +1007
```

As a decimal number:

```
1.09 2300.18 -2.178262
```

In scientific notation:

```
12E7 6.028E-23 -0.928E+18
```

As a hexadecimal integer:

```
&H14 &H3B1E &HFFFC
```

Numbers written without either a decimal point or a power of ten which are in the inclusive range -32768 to 32767 will be treated as integers by Marinchip BASIC. Marinchip BASIC will automatically perform integer calculations unless forced to floating point by an overflow or appearance of a fraction. Since integer arithmetic is up to 500 times faster than floating point arithmetic, the user should leave out decimal points when they are not required.

When writing numbers in scientific notation, the number following the "E" indicates a power of ten by which the number preceding the "E" is multiplied. For example, one million (1000000) may be written 1E6, and

one millionth (0.000001) may be written 1E-6. This allows very large or very small numbers to be written compactly.

Numbers in Marinchip BASIC must fall between 8.638E-78 and 7.237E75. Any number outside this range, whether explicitly specified or generated as an intermediate result, will cause an overflow error. Marinchip BASIC (B) maintains between six and seven digits of internal accuracy for all numbers. Extended Commercial BASIC (EB) and Transaction BASIC (TB) maintain sixteen digits of internal accuracy for all numbers. All arithmetic is performed with additional internal precision to guard against round-off errors.

A hexadecimal integer may be written by preceding the hexadecimal digits with the characters "&H". The hexadecimal integer must be greater than or equal to zero (&H0) and less than or equal to &HFFFF.

2.4.2. String constants

Strings are made up of a sequence of zero or more characters. String constants are written by enclosing the characters in quote marks. Either the single quote (') or the double quote (") may be used, but the quotes on both ends of the string must be the same kind. Within a string enclosed in one type of quote, the other type of quote may appear, for example, the following are valid strings:

```
"doesn't"  'It was 7" long.'
```

A quote of the same kind used to enclose the string may be included in the string by writing two quotes in a row. The two quotes will cause one quote to be considered as part of the text of the string. For example:

```
'ain''t'  """"
```

The last example represents a string consisting of one double quote mark. The first quote identifies the start of the string, the second and third represent the quote character, and the fourth identifies the end of the string.

2.5. Variables

BASIC variables are used to hold values that may change during the execution of a program. Variables may be either numeric (which hold numbers), or string (which hold groups of characters) and may be either simple, which hold a single value, or array, which hold multiple values. All combinations are possible and meaningful, e.g.,

```
Simple numeric
Simple string
Subscripted numeric
Subscripted string
```

2.5.1. Numeric variables

Numeric variables are composed of either a letter (A through Z), or a letter followed by a digit (0 through 9). For example, some simple numeric variables are:

```
A Q9 V7
```

Numeric variables may optionally be written as a letter followed by one or more letters or numbers. Such long variable names may be of any length, but only the first two characters are used to distinguish different variables (e.g., BOMB and BOOK are the same variable). Long variable names may not contain a BASIC keyword (statement, function name, or command) anywhere within them. Examples of long variable names are:

```
BLUE TURKEY Z9M9Z SIGNAL5
```

Note that "CARTON" would not be permissible, since it contains the BASIC keyword "TO".

2.5.2. String variables

String variables are composed of either a letter followed by a dollar sign ($), or a letter followed by a digit followed by a dollar sign. Some typical simple string variables are:

A$ B5$ Z6$

String variables may also be written as a letter followed by one or more letters or numbers as described above for numeric variables. The entire variable name is followed by a dollar sign. The restrictions on embedded BASIC keywords apply to string variables also. Examples of long string variable names are:

NAME$ ADDRESS$ ARGLEBARGLE$

2.5.3. Subscripted variables

A subscripted variable represents a vector or matrix which can store multiple values. The number of subscripts used with a variable and their maximum values is fixed when the subscripted variable is declared via a DIM statement. All subscripted variables must be declared by a DIM statement before use: there is no automatic declaration. The lowest value of a subscript is always zero, and the highest value is specified in the DIM statement. For example, to declare a list of numbers whose subscript may range from zero to 10, one would use:

```
DIM N(10)
```

Then, to reference element 5 in this list of numbers, one would write:

```
N(5)
```

Subscripted variables may be strings as well as numbers. In the case of a string, a dollar sign follows the variable name and precedes the parenthesis. For example:

```
DIM F1$(20)
F1$(18)
```

Subscripted variables may have from one to ten subscripts. While all of these examples have used constants as subscripts for clarity, in fact, a subscript may be any numeric expression within the bounds of 0 to the limit given in the DIM statement. If the subscript has a fractional part (e.g., 3.8) the fraction will be truncated. For example V(12.3) selects element 12 of the variable V.

2.5.4. Distinctness of variables

String variables and numeric variables with the same name are permitted and are distinct, and simple and subscripted variables with the same name are distinct. As a result, the following are four completely different and unrelated variables:

```
A          - A simple numeric variable
A(7)       - Element 7 of a numeric array
A$         - A simple string variable
A$(7)      - Element 7 of a string array
```

Note that extensive use of this feature may make programs very confusing and hard to follow.

2.5.5. Declaration of variables

A subscripted variable must be declared by the execution of a DIM statement before any element of the variable is used. A simple variable is declared when a value is first assigned to it by the execution of a LET statement, or by being read in by a READ or INPUT statement. String variables may also be created with the MID$ statement (see below). If a variable is used in an expression before it has been given a value, the program will error with an "Undefined variable" message.

2.6. Operators

Operators are special characters or words that indicate operations to be performed on variables, constants, or other values.

2.6.1. Arithmetic operators

The arithmetic operators are as follows:

```
( )          - Parentheses to group items
^ or **      - Exponentiation
-            - Unary minus (e.g., -A)
*  /         - Multiply and Divide
\            - Integer divide (remainder discarded)
MOD          - Modulo (remainder from divide)
+ -          - Add (concatenate) and Subtract
```

The arithmetic operations are performed in the order listed above. Note that since parentheses have the highest priority, parenthesised expressions will always be evaluated first. For example, the expression:

```
A+B*C
```

means to multiply B and C, then add A. Since multiplication has a higher priority than addition, it is performed first. If the expression were written:

```
(A+B)*C
```

A and B would be added, and the result multiplied by C. The use of parentheses forces the computation of A+B first. Operations with the same priority are performed left to right. While a user familiar with the operator priority rules can write a very complicated expression without any parentheses, the program may be more readable if the theoretically redundant parentheses are included. For example:

```
C/B**-F/B9          is correct
```

but:

```
(C/(B**(-F)))/B9    is better
```

The addition operator (+) may also be used with two strings to concatenate string consisting of the left operand string with the right operand string appended to the end. For example, if the string variable X$ contains the string "Chuck", the expression:

```
"Hello there, "+X$+"!"
```

would have the value "Hello there, Chuck!". None of the other arithmetic operators (except parentheses) may be used with strings.

2.6.2. Relational operators

The relational operators are:

```
=            - Equality
<> or ><     - Inequality
```

```
<           - Less than
>           - Greater than
<= or =<    - Less than or equal
>= or =>    - Greater than or equal
```

The relational operators are used to determine the relation between two values. The relational operators may be used with both numeric values and strings, but may not be used to compare numbers to strings, and vice versa. When comparing numbers, their numeric values, including sign, are compared. When comparing strings, the strings are examined from left to right for the first differing characters, and the relative magnitudes are determined from the ASCII collating sequence of the differing characters. If one string is shorter than the other, but both are equal up to the end of the shorter string, the shorter string is considered to be the lesser.

The value returned by the relational operators is always numeric, and equal to -1 if the relation is true and 0 if the relation is false. All relational operators have the same priority, so they are executed from left to right. Relational operators have a priority less than any arithmetic operator, so the expression:

```
A+B<C+9
```

will test whether A+B is less than C+9, as expected.

2.6.3. Logical operators

The logical operators are as follows:

```
NOT         - Logical negation (unary)
AND         - Logical conjunction
OR          - Logical disjunction
XOR         - Logical exclusive or
EQV         - Logical equivalence
IMP         - Logical implication
```

The logical operators operate bit by bit on the integer part of their operands. When used on values returned by the relational operators, the logical operations take on their conventional definitions with -1 defined as TRUE and 0 defined as FALSE. In addition, the logical operators may be used to perform bit by bit masking operations. The operands used with a logical operator are converted to integer before the operation is executed, and if the operands are outside the range -32768 to 32767 an overflow will occur. The logical operators may not be used with strings. The priority of the logical operators are in the order listed above, and all logical operators have a priority below that of any relational operator. Hence, the expression:

```
A+B>D/2 OR NOT F<G/9
```

will be interpreted as:

```
((A+B)>(D/2)) OR (NOT (F<(G/9)))
```

2.7. Functions

Functions take a set of values as ARGUMENTS and return a value as a RESULT. A function is identified by a name. Functions may be either internal functions, provided by BASIC, or user functions, written by the programmer and defined in the BASIC program. Some functions return numeric values, while others return strings. All functions which return string results have names which end in a dollar sign.

2.7.1. Numeric internal functions

Numeric functions are those that return a number as their result. Note that some numeric functions accept strings as arguments. The mathematical functions ATN, COS, EXP, LOG, SIN, SQR, and TAN return results accurate to approximately six decimal digits regardless of which version of BASIC is

being used.

2.7.1.1. ABS(<i>) - Absolute value

The value returned is the absolute value of the numeric argument <i>. For example, ABS(12.9) is 12.9, and ABS(-18.2) is 18.2.

2.7.1.2. ASC(<x$>) - ASCII code

The value returned is the ASCII code for the first character of the string argument <x$>. If the string argument is of zero length, the ASCII code for space will be returned.

2.7.1.3. ATN(<i>) - Arctangent

The value returned will be the arctangent in radians of the value <i>.

2.7.1.4. CINT(<i>) - Convert to integer

The value returned will be <i> with any fractional part discarded. If <i> is outside the range -32768 to 32767, an overflow will occur.

2.7.1.5. CSNG(<i>) - Convert to single precision

The value returned will be <i> converted to floating point.

2.7.1.6. CDBL(<i>) - Convert to double precision

The value returned will be <i> converted to floating point. In Marinchip BASIC CSNG and CDBL always perform the same function. In B, the conversion is always to single precision. In EB and TB, conversion is always to double precision.

2.7.1.7. COS(<i>) - Cosine

radians. assuming <i> is in

2.7.1.8. CVD(<x$>) - Convert to double

The leftmost eight characters of string <x$> are directly converted to a double precision real number. In other words, the binary value of the characters is taken as the floating point number. This function is used to convert back strings returned by the MKD$ function (see below). If used in simple BASIC, CVD will return a single precision number.

2.7.1.9. CVI(<x$>) - Convert to integer

The leftmost two characters of string <x$> are directly converted to an integer. This function is used to convert back string data packed using the MKI$ function (see below).

2.7.1.10. CVS(<x$>) - Convert to single precision

The leftmost four characters of string <x$> are directly converted to a single-precision floating point number. This function is used to convert back string data packed using the MKS$ function (see below). If used in EB or TB, this function will produce a double precision floating point number, but only six to seven digits of accuracy will be preserved.

2.7.1.11. EXP(<i>) - E to the power

The value returned will be the mathematical constant E raised to the power <i>.

2.7.1.12. EOF(<i>) - Test end of file

The EOF function returns zero if file number <i> is not at end of file, and -1 if it is at end of file. This function is used in conjuction with the file I/O statements to determine when the last record of a file has been read.

2.7.1.13. FIX(<i>) - Convert to integer

The result will be the value of <i> with any fractional part removed. This function differs from INT in that negative numbers will not be rounded down. FIX(a) is equivalent to:

SGN(a)*INT(ABS(a)).

2.7.1.14. INP(<i>) - Input

The value returned is an integer from 0 to 255 representing the value that resulted from reading machine input port <i>. <i> must evaluate to a number from 0 to 255.

2.7.1.15. INSTR(<i>,<x$>,<y$>) - In string

The string <x$> is searched starting at character <i> for the string <y$>. If <y$> is not found, zero is returned, otherwise the character at which the matching substring starts is returned. If <i> is omitted [in other words, the call is INSTR(<x$>,<y$>)] the search will start at the first character of <x$>.

2.7.1.16. INT(<i>) - Largest integer

The fractional part (if any) of <i> is discarded. In other words, the largest integer in <i> is returned.

2.7.1.17. LEN(<x$>) - Length

The number of characters in the string <x$> will be returned.

2.7.1.18. LOC(<i>) - Location in Direct file

The LOC function will return the number of the next record to be read or written in Direct access file <i>. See the section on Direct access files for more information.

2.7.1.19. LOG(<i>) - Natural logarithm

The natural (base E) logarithm of <i> is returned.

2.7.1.20. PEEK(<i>) - Examine memory

The byte at absolute memory address <i> is returned. The argument <i> may have a value between 0 and 65535. Negative values greater than or equal to -32768 are also accepted, and address bytes with the equivalent 16 bit two's complement addresses. Note that PEEK returns a byte value between 0 and 255. To load a whole word, two PEEKs must be done.

2.7.1.21. RND(<i>) - Random

A pseudo-random number greater than or equal to 0 and less than 1 is returned. The argument <i> is ignored and may be omitted. If omitted, the parentheses must be omitted also.

2.7.1.22. SGN(<i>) - Sign

The sign of the numeric argument <i> is examined. If <i> is equal to zero, 0 is returned. If <i> is positive, 1 is returned; and if <i> is negative, -1 is returned.

2.7.1.23. SIN(<i>) - Sine

The sine of the angle <i> specified in radians is returned.

2.7.1.24. SQR(<i>) - Square root

The square root of the argument <i> is returned. If <i> is negative the program will be errored.

2.7.1.25. TAB(<i>) - Tabulation

The TAB function may be used only within a PRINT statement. It causes the output column to be set to the integer part of its argument. The argument must be less than or equal to the current output line width (set by the WIDTH statement). The leftmost column on the output line is column zero.

2.7.1.26. TAN(<i>) - Tangent

The tangent of the angle <i> in radians is returned.

2.7.1.27. VAL(<x$>) - Value

The string <x$> is evaluated as a numeric constant according to the syntax rules for BASIC numbers. The numeric result is returned.

2.7.2. String internal functions

The string functions return string results. Note that some string functions take numeric arguments.

2.7.2.1. CHR$(<i>) - Character

The CHR$ function returns a one character string which has the character with the ASCII code equal to the expression <i>. If the value of <i> is outside the range -32768 to 32767, an overflow will occur. If <i> is outside the range 0 to 127, the upper 8 bits will be removed to reduce it to within this range.

2.7.2.2. HEX$(<i>) - Hexadecimal

The value returned is the integer part of the argument <i> edited as four hexadecimal digits. If the argument <i> is outside the range -32768 to 32767 an overflow will occur.

2.7.2.3. LEFT$(<x$>,<i>) - Leftmost part

The leftmost <i> characters of the string value <x$> will be returned. If <i> is greater than the number of characters in the string <x$>, all of <x$> will be returned.

2.7.2.4. LWR$(<x$>) - Lower case

The string <x$> will be returned with all alphabetic characters converted to lower case.

2.7.2.5. MID$(<x$>,<i>,<j>) - Extract substring

The <j> characters starting with character <i> will be extracted from string <x$> and returned. If all or part of the substring to be extracted is outside the length of <x$>, spaces will be returned for the out of bounds characters. If <j> is omitted (the function is called with two arguments), the rightmost <i> characters of <x$> will be returned.

2.7.2.6. MKD$(<i>) - Make double precision string

The binary representation of the numeric value <i> is returned as an eight character string. This function is most often used in packing string records used with Direct access files. Since any ASCII character may be a part of the string returned by this function, caution must be exercised in using these strings in sequential files where control characters may cause improper interpretation of data. This function is normally used in programs written in EB and TB, but may be used as well in B. In B, the function will still return eight characters, but only the first four will be significant, as B implements only single precision numbers.

2.7.2.7. MKI$(<i>) - Make integer string

The numeric value <i> is truncated and converted to an integer. If the value is outside the range -32768 to 32767 an overflow will occur. A two character string is returned whose characters consist of the binary representation of the integer result. The caution given above for MKD$-generated strings applies to strings generated by MKI$ as well.

2.7.2.8. MKS$(<i>) - Make single precision string

The binary representation of the numeric value <i> is returned as a four character string. When used in B, this function will preserve the complete accuracy of a number. In EB and TB, use of this function and its inverse CVS will cause the accuracy of a number to be truncated to six to seven decimal places. The caution given above about the use of MKD$

strings in sequential files applies to strings generated by MKS$ as well.

2.7.2.9. RIGHT$(<x$>,<i>) - Rightmost part

The <i> rightmost characters of the string value <x$> will be returned. If the value of <i> exceeds the length of <x$>, all of <x$> will be returned.

2.7.2.10. SPACE$(<i>) - Spaces

A string consisting of <i> blanks will be returned.

2.7.2.11. STR$(<i>) - String representation

The value <i> will be edited and returned as a string. The string returned will be identical to what would be printed if <i> were placed on a PRINT statement.

2.7.2.12. UPR$(<x$>) - Upper case

The string <x$> will be returned with all alphabetic characters converted to upper case.

2.7.3. User functions

User functions are declared by the programmer via the DEF statement, and have names of the form FNx where X is a letter from A to Z. The declaration of a function is of the form:

DEF FN<letter>[$][(<formal arg list>)]=<expression>

where <letter> is the letter from A to Z identifying the function, <formal arg list> is an optional list of variables to represent the arguments with which the function is called, and <expression> is an arbitrary expression which computes the value of the function. For example, the following declaration will define a function which computes the average of two numbers:

```
160 DEF FNA(A,B) = (A+B)/2
```

Once defined by the execution of the DEF statement, the function is used like any BASIC internal function. For example, to set the variable F equal to the average of the variables Q and W, one would write:

```
230 F=FNA(Q,W)
```

Note that the variables used in the <formal arg list> and in the expression which defines the function have no existence outside the function: they are used only to represent the arguments with which the function is called. Other program variables may be used in the function expression. If a variable used in the function expression is defined both in the <formal arg list> and elsewhere in the program, the <formal arg list> variable will be used. User function arguments may be either numeric or string, but the type of the argument with which the function is called must agree with the type of the variable used in the <formal arg list> when the function was declared. User functions may return either string or numeric values. User functions which return strings should be declared and called with a dollar sign ($) following the function name. User string functions and numeric functions are distinct: the functions FNC and FNC$ may be defined at the same time. For example, the following is the declaration and use of a function that takes a string and a numeric argument and returns a string consisting of the string argument with the numeric argument appended.

```
560 DEF FNE$(S$,N)=S$+" "+STR$(N)
```

```
860 PRINT FNE$("Error in line",L1)
```

The <formal arg list> is optional. Functions may be declared which have no arguments, in which the expression references only program variables and constants. Such functions are a convenient shorthand for lengthy expressions used frequently in a program. For example:

```
750 DEF FNS=X+1/Y+INT(Z+COS(T5))+FNA(R,8)
980 V(FNS)=V(FNS)+G(FNS)
```

A function is declared by the execution of the DEF statement, hence the DEF statement must be executed before the first reference to the function in the program. Functions may be redeclared as frequently as desired within a program, simply by executing another DEF statement referencing an existing function name with a different <formal arg list> and/or <expression>.

3. Statements

The following paragraphs describe the BASIC statements. Every line in a BASIC program is a statement of some kind. Each statement is preceded by a line number. Line numbers may be assigned freely by the user, but must be greater than or equal to 0 and less than or equal to 32767. Statements are normally executed in order of ascending line number, and are always printed out and written to files in line number order. In addition, line numbers are used to identify the destination of GO TO and GO SUB statements.

Multiple statements may be written on one line by separating them by colons (:). Whenever BASIC would normally execute the next statement in the program, it will check whether a colon follows the statement just completed. If so, execution will continue following the colon rather than on the next line. Examples of statements written using this feature are:

```
110 A=1: B=1: C$='Hello'
210 IF A=1 THEN PRINT "It was 1": GO TO 180 ELSE 605
310 FOR I=1 TO 100 : PRINT I,SIN(I) : NEXT I
```

3.1. Comment statement

3.1.1. REM

Any line beginning with the letters "REM" will be treated as a comment and totally ignored by BASIC. Any characters following the REM will be totally ignored and can be used to add comments to a program. Examples of comments are:

```
100 REM  This program computes the square root of zero
120 REMARKABLE PROGRAM!
130 REMNANT OF A FORGOTTEN AGE
```

3.2. Data transfer statements

3.2.1. LET

```
LET <variable>=<expression>
```

The LET statement is used to assign the value of an expression to a variable. The variable and expression may be either numeric or string, but must be of the same type. The variable may be subscripted. The word LET may be omitted, if desired. Hence, the following are all valid LET statements:

```
100 LET A=19
110 A7=12.8+B3
120 LET V9(18,Q+4)=C3(INT(Q+9))
130 LET A$="This is my string"
140 A$(Q,B+3)=LEFT$(M$,3)
```

Be careful to distinguish the equal sign in the LET statement, which means "becomes", from the relational operator "=" which means "is equal to". The statement:

```
820 LET V=R=1
```

sets V equal to -1 if R equals 1, and 0 otherwise.

3.2.2. MID$

MID$(<string var>,<start>[,<length>])=<string expression>

The MID$ statement allows replacement of a substring of a string variable. The substring starting at the character indicated by <start>, and extending for <length> characters within the variable <string var> will be set to the first <length> characters of <string expression>. If the <length> specification is omitted, the length of <string expression> will be used. The <string var> will be made longer if necessary to contain the selected substring. Assume that variable A$ contains the string:

"This is a good program."

then the statement:

165 MID$(A$,11,4)="poor choice!"

will set A$ equal to:

"This is a poor program."

3.2.3. SWAP

SWAP <variable>,<variable>

The values assigned to the two variables are interchanged. The variables may be either numeric or string, but must be the same type. This permits values to be interchanged without the requirement for an intermediate variable. If X equals 19 and Y equals 104, the statement:

208 SWAP X,Y

will assign 104 to X and 19 to Y.

3.3. Control statements

In the absence of control statements, statements in a program are executed in order of ascending line number. Control statements permit the user to alter the flow of control and create loops and decision branches within a program.

3.3.1. GO TO

GO TO <statement number>

The GO TO statement causes the line with <statement number> to be executed next.

3.3.2. GO SUB

GO SUB <statement number>

The GO SUB statement causes the line with <statement number> to be executed next. The statement following the GO SUB is saved in memory so that it may be returned to by a RETURN statement. GO SUB statements may be nested (that is, part of a program called by a GO SUB may call other parts with GO SUB). The only limit on nesting is the amount of available memory to save the return points. GO SUB works correctly with multiple statements on a line, permitting the following:

323 PRINT "Hi" : GO SUB 1800 : PRINT "We got back!"

3.3.3. RETURN

```
RETURN
```

The next statement executed will be the statement following the most recent GO SUB statement.

3.3.4. ON

```
ON <expression> GO TO <statement no>,<statement no>,...
ON <expression> GO SUB <statement no>,<statement no>,...
```

The <expression> is evaluated, any fractional part is discarded, and the integer value is used to select one of the statement numbers following the GO TO or GO SUB. If the <expression> is outside the range -32768 to 32767 an overflow will occur. If the <expression> is negative, zero, or larger than the number of <statement no> specifications, the ON statement will be ignored and the next statement will be executed. Otherwise, a GO TO or GO SUB (depending on which is used) will be performed to the selected <statement no>. For example, if J was equal to 3, the statement:

```
305 ON J GO TO 150,180,220,105,130
```

would cause statement 220 to be executed next.

3.3.5. FOR

```
FOR <variable>=<start> TO <limit>
FOR <variable>=<start> TO <limit> STEP <step>
```

The <variable> is set equal to the numeric expression <start>. Then the <limit> and, if specified, <step> expressions are evaluated and saved. If <step> is omitted, it is assumed to be 1. The block of statements between the FOR and the next NEXT statement with the same <variable> will be executed repeatedly, each time adding <step> to <variable> until the <limit> is exceeded. For example, to print the numbers 1 through 10 and their squares, the following might be used:

```
100 FOR I=1 TO 10
110 PRINT I, I*I
120 NEXT I
```

to print the same table in descending order, one might use the following:

```
100 FOR I=10 TO 1 STEP -1
110 PRINT I,I*I
120 NEXT I
```

Note that the statements in a FOR loop will always be executed at least once, even if the <start> expression is already beyond the <limit>. Also, note that the NEXT statement must be executed in order to perform the test against the <limit> and possibly transfer control to the top of the loop. Hence, a NEXT may be the object of an IF statement. Programs which use this feature are generally very hard to understand.

3.3.6. NEXT

```
NEXT <variable>
```

Encountering a NEXT statement specifying the <variable> of an active FOR loop causes the <step> to be added to the <variable> and the <variable> to be tested against the <limit> of the loop. If the <limit> is exceeded, control continues following the NEXT. Otherwise, control resumes following the FOR statement for <variable>. See the FOR example above for an illustration of the use of NEXT with FOR. NEXT may be used with a list of variables separated by commas as a shorthand way to terminate multiple nested FOR loops. For example, the following program which clears a three dimensional array:

```
200 FOR I=1 TO 10
210 FOR J=1 TO 10
220 FOR K=1 TO 10
230 A(I,J,K)=0
240 NEXT K
250 NEXT J
260 NEXT I
```

may be rewritten:

```
200 FOR I=1 TO 10
210 FOR J=1 TO 10
220 FOR K=1 TO 10
230 A(I,J,K)=0
240 NEXT K,J,I
```

3.3.7. IF

```
IF <expression> THEN <statement no>
IF <expression> GO TO <statement no>
IF <expression> THEN <statement>
IF <expression> THEN <statement> ELSE <statement>
```

The IF statement allows conditional execution of statements and transfer within a program depending upon the value of expressions. The numeric <expression> will be evaluated. If nonzero, the item following the IF will be executed. If zero, the statement will be ignored unless an ELSE is specified. In that case, the item following the ELSE will be executed. An IF ... THEN ... ELSE statement will accept either statement numbers or regular statements following the THEN and ELSE. If a statement number is specified, control will be transfered to that statement. If a statement is specified, that statement will be executed, and control will resume following the IF statement. To illustrate the many forms of the IF statement, some examples follow:

```
IF A=1 THEN 290              - Goes to 290 if A=1
IF A=1 THEN 290 ELSE 650     - Goes to 290 if A=1,
                               650 otherwise.
IF A=1 GO TO 290             - Goes to 290 if A=1
IF A=1 THEN PRINT "Ok"       - Prints "Ok" if A=1
IF A=1 THEN PRINT "Ok" ELSE 350
                             - Prints "Ok" if A=1
                               goes to 350 otherwise.
IF A=1 THEN PRINT "Ok" ELSE PRINT "Bad"
                             - Prints "Ok" if A=1,
                               "Bad" otherwise.
```

Since any statement may be used after THEN or ELSE, including another IF, there is a potential ambiguity in associating an ELSE with an IF in such a statement as:

```
IF A>4 THEN IF A>8 THEN 280 ELSE 300
```

the ambiguity is resolved by always grouping an ELSE with the immediately preceding IF.

By using the feature of separating multiple statements with a colon, it is possible to have several statements after a THEN or ELSE. For example:

```
IF A<10 THEN A=A+1 : PRINT A ELSE PRINT "End." : GO TO 100
```

3.3.8. END

```
END
```

The END statement terminates the execution of the BASIC program, closes all open files, and returns control to the command handler. The END statement is normally the last statement in a BASIC program.

3.3.9. STOP

STOP

The STOP statement is identical to the END statement described above. For compatibility with other BASIC systems, there should only be one END statement in a program, and it should be last. The STOP statement should be used to terminate the execution of a program from within the middle of the program. Alternately, the program may GO TO the END line.

3.3.10. PAUSE

PAUSE

The PAUSE statement suspends execution of the BASIC program. Variables continue to keep their values, files remain open, and control returns to the command handler. This statement allows the program to permit the user to examine it, modify it, and later resume execution with the CONTINUE command.

3.4. Array statements

The array statements allow the declaration and release of arrays (dimensioned variables). Use of these statements can allow dynamic allocation and release of storage in a BASIC program.

3.4.1. DIM

DIM <variable>(<expression>,...),...

The DIM statement declares dimensioned variables and assigns storage for them. The variable name is followed by a dimension list enclosed in parentheses. One entry must occur for each subscript to be used with the variable. The entries declared are from zero to the value of the expression. The statement:

```
450 DIM A(10),B7$(5,80)
```

declares two arrays. The variable A is a numeric array with elements A(0) to A(10). The variable B7$ is a two dimensional string array with elements B7$(0,0) to B7$(5,80). Note that the dimension limits can be expressions as well as the numbers used in the example above. This permits, for example, reading in the number of elements in a file from a record in the file, creating an array with that number of elements, then reading the file into the array. It is important to remember that DIM is an executable statement: it must be executed in order to create the array variable. If a variable has been created by a DIM statement, an attempt to create it again will cause the program to error with the message:

Attempt to redimension variable in <line>.

An array can be deliberately redimensioned by destroying it with the ERASE statement (see below), and then creating it with the new dimensions via another DIM statement.

3.4.2. ERASE

ERASE <variable>,<variable>,...

The ERASE statement releases the storage assigned to an array variable. When an array variable is declared by the execution of a DIM statement, storage is assigned to hold the elements of the array. When this storage is no longer required, the program can release it by performing an ERASE on the variables. For example:

```
620 ERASE A,B7$
```

After an ERASE, the variables can be re-created with a DIM statement, with any desired bounds or number of subscripts.

3.5. Function statement

3.5.1. DEF

DEF FN<letter>[$][(<formal arg list>)]=<expression>

The DEF statement defines a user function by specifying the function name (always "FN" followed by a single letter), an optional list of formal arguments, and an expression that computes the value of the function. The DEF statement must be executed to define the function before the function is first used in the program. A function may be redefined by execution of subsequent DEF statements. For more information on the use of DEF and user functions, refer to the section on "User functions" in chapter 2 of this manual.

3.6. Data input statements

This section describes the READ, DATA, and RESTORE statements, which permit reading of data supplied by statements within the program. This facility permits easy initialisation of variables as well as access to constants required repetitively during the execution of a program.

3.6.1. READ

READ <variable>,<variable>,...

The READ statement reads successive items from the DATA statements in the program and assigns them to the <variable>s on the READ statement. All of the items appearing on DATA statements in the program (see below) are taken as a list. Each <variable> in a READ statement is assigned the next data item. If the <variable> is a numeric variable, the data item must be a number, and if the <variable> is a string, the data item must be a string. The data is read without regard to its distribution between separate DATA statements. The <variable>s used on the READ statement may be subscripted. If a READ statement attempts to read past the end of the DATA present in the program, an "Out of data" error message will be issued.

3.6.2. DATA

DATA <constant>,<constant>,...

The DATA statement, which is not executed, lists data items which are read by the READ statement. Data appearing in a program are read in order of their appearance (left to right within a DATA statement, and in order of ascending line numbers on DATA statements) with each data item being assigned to each <variable> appearing on a READ statement. A DATA statement may contain one or more constants; the constants are separated by commas. The constants may be either BASIC numeric or string constants. If a constant is found that begins with a letter, it will be scanned as a string terminated by a comma. Such a specification will have all spaces compressed out, so string constants on a DATA statement which require embedded spaces or commas must be quoted. The following is an example of a DATA statement:

```
480 DATA 12,19,2.76,25.7E-8,"Results 1",good,average,bad
```

3.6.3. RESTORE

```
RESTORE
```

The RESTORE statement resets the pointer to the DATA in a program to the first item on the lowest numbered DATA statement. That is, it restores the data to the state at the beginning of the program execution. This statement is used when a program desires to repeatedly read the data on DATA statements within the program.

3.7. Interactive I/O statements

The INPUT, LINE INPUT, PRINT, PRINT USING, and WIDTH statements allow BASIC to communicate with the system terminal. This allows the user to input to the program and to receive results as the program runs.

3.7.1. INPUT

```
INPUT <variable>,<variable>,...
INPUT <string>;<variable>,<variable>,...
```

The INPUT statement accepts constants from the terminal and assigns their values to program variables. In the first form of the statement, the user will be prompted with a question mark. In the second form, the <string> will be typed as the prompt for the input. If multiple <variable>s are specified on the INPUT statement, the multiple data items entered from the terminal must be separated by commas. If the <variable> on the INPUT statement is a numeric variable, a number must be entered on the terminal, and if the <variable> on the INPUT statement is a string, a string must be entered. A string starting with a letter may be entered without enclosing it in quotes, but if a string is to have embedded blanks or spaces, quotes must be used. Any error in the data entered to the INPUT statement will cause the message:

```
Bad data.  Re-enter from start.
```

to be issued and the prompt to be repeated. The user must then retype ALL the items expected by the INPUT statement. If fewer items are entered than there are <variable>s on the INPUT statement, a question mark prompt will appear and the user will be expected to supply the rest of the input expected. The user will continue to be prompted until all the <variable>s on the INPUT statement have been satisfied. Examples of INPUT statements are:

```
INPUT L
INPUT I,J,K,L
INPUT "Enter next data point: ";X,Y
INPUT "Type your name: ";N$
```

3.7.2. LINE INPUT

```
LINE INPUT <string variable>
LINE INPUT <string>;<string variable>
```

The LINE INPUT statement permits a line of input from the keyboard to be read transparently and stored, exactly as entered, in a string variable. If the first form of the LINE INPUT statement is used, no prompt at all will be typed, while the second form will cause the <string> specified to be typed as the prompt for the line of input. Note that only one <string variable> may be read by a LINE INPUT statement. For example, to read a line of input into the variable A$:

```
650 LINE INPUT "Enter next line: ";A$
```

3.7.3. PRINT

PRINT <expression>,<expression>;<expression>...

The PRINT statement displays the value of both numeric and string expressions on the interactive terminal. The simplest form of the PRINT statement is simply:

PRINT

which will cause a blank line to be printed. If a variable name is supplied, it will be edited in a format depending upon its type and value. A number will be edited in either integer, decimal, or exponential form depending upon its value and magnitude, and a string will be copied to the output line. Either commas or semicolons may be used to separate multiple expressions on a PRINT statement. A comma causes the next expression to be printed at the start of the next 14 character field, while a semicolon causes the next item to start immediately after the last one. If the last item on a PRINT statement is an expression, the carriage will be returned after printing the last value. If a comma or a seimcolon is the last item on a PRINT statement, the carriage will remain extended following the execution of the PRINT. This can be used to allow an output line to be built up through the execution of multiple PRINT statements, or to type a prompt for a subsequent INPUT statement. For example:

```
420 PRINT "Enter your name: ";
440 INPUT N$
```

Output generated by the PRINT statement may be aligned to any column by the TAB function. If an <expression> supplied to PRINT consists solely of a call on the TAB function, the output will be tabulated to that column rather than printing any value. The leftmost column on the output line is column zero. The TAB function is normally followed by a semicolon to cause the next value to start in the column selected by the TAB function. For example, to print three numbers in columns 10, 20, and 30, you might use:

```
520 PRINT TAB(10); I; TAB(20); J; TAB(30); K
```

If the output generated by one PRINT statement is longer than the line on the output device, it will be continued onto as many lines as are required. The length of the output device is set by the WIDTH statement (see below).

3.7.4. PRINT USING

PRINT USING <format>,<expression>,...

The PRINT USING statement prints an output line formatted according to the specifications of a <format>, which is a general string expression appearing as the first item on the PRINT USING statement. Characters in the <format> string will be copied to the output, except that special sequences of characters are replaced by edited versions of the expressions appearing on the PRINT USING statement. The editing performed is specified by the special characters in the <format>. Because the editing specifications are usually a generic representation of the output desired, they are referred to as the "picture" of the output field. Any character not listed below as a picture character will be copied directly to the output. The PRINT USING statement will reprocess the <format> string until all <expression>s have been edited. Hence, if an output line is to have all <expression>s edited with the same format, it need only specify one number in the format.

PRINT USING is available only in EB and TB.

3.7.4.1. ! - Single character string picture

The ! character in the <format> string will be replaced by the first character of the next <expression> in the list on the PRINT USING statement. If the next <expression> is numeric, an error will occur.

3.7.4.2. \ \ - Multiple character string picture

The two backslashes and the characters between them will be replaced by the same number of characters starting at the beginning of the next <expression> (which must be a string). If the <expression> string is not as long as the field is wide, the string will be padded with spaces on the right to fill the field.

3.7.4.3. & - Variable length string picture

The & character will be replaced with the value of the next <expression> (which must be a string). The entire string value will be inserted into the output line.

3.7.4.4. # - Numeric picture

The # character begins a numeric field. The number sign characters represent digits, and the field may contain as many digits as desired before and after the decimal point. If no decimal point appears in the picture, the decimal digits will not be edited. If a minus sign (-) appears immediately following a numeric field, a minus sign will be edited following the number if it is negative, and a space will be edited following it if it is positive. Otherwise, the minus sign will be printed before the number if it is negative. If a comma appears between two # characters before the decimal point, the number will be edited with commas between each group of three digits. The commas count as places in the number, so they should be written where commas are desired in the number (although commas will always be edited correctly regardless of where the commas were specified in the picture). Examples of simple numeric fields are:

```
#                   - Single digit integer
####.##             - Decimal number
##,###,###.##       - Decimal number with commas
######.###-         - Decimal number with trailing sign
```

If the value of the <expression> is too large to represent in the field size given, the field will be filled with asterisks.

3.7.4.5. ** - Check protected numeric picture

A field beginning with two asterisks is a numeric field which will be printed with all positions before the decimal place not used for the number filled with asterisks. This feature is called "check protection", and is normally used to prevent alteration of numbers printed on negotiable instruments. A field beginning with two asterisks may continue like any other numeric field, with either asterisks or number signs (#) used to denote digits. The following are examples of check protected fields:

```
****.**             - Simple decimal check protected field
****.##             - Same as above
**,***,***.**       - With commas
*****-              - With trailing sign
```

Note that two asterisks must appear in a row to denote a check protected field. A single asterisk will simply be printed in the output like a non-picture character.

3.7.4.6. $$ - Float dollar sign numeric picture

A field beginning with two dollar signs is a numeric field in which a dollar sign will be printed to the left of the most significant digit of the number. Following the two dollar signs is a normal numeric picture as described for the # picture character. All the options applicable for # numeric fields may be used with $ fields. Note that two dollar signs must

appear in a row to denote a picture field. A single dollar sign will simply be copied to the output. Examples of float dollar sign fields are:

```
$$              - A two character field
$$###.##        - Five before decimal, two after
```

3.7.4.7. **$ - Check protect and float dollar sign

If a field begins with two asterisks and a dollar sign, it denotes both floating dollar sign and check protection. Because of the notation used for such a field, all such fields must be at least three characters wide. Examples of such fields are:

```
**$,***,***.**  - Lots of money
**$#            - Four character float, protect
```

3.7.4.8. Examples of PRINT USING pictures

The following program fragment illustrates the use of PRINT USING to produce formatted output.

```
1200 PRINT "  Part            In Stock        Cost"
1210 F$ =  "\               \      ######     $$###.##"
1220 V=0
1230 FOR I=1 TO N
1240 PRINT USING F$,P$(I),S(I),C(I)
1250 V=V+C(I)*S(I)
1260 NEXT I
1270 PRINT USING "Total inventory value $$#####.##",V
```

3.7.5. WIDTH

```
WIDTH <expression>
```

The WIDTH statement sets the interactive output line width to the integer part of the <expression>. If the <expression> is outside the inclusive range 1 to 132, an overflow will occur. The PRINT statement will continue output that exceeds the current WIDTH setting onto multiple lines. The default WIDTH setting is 80 characters. WIDTH affects only interactive output; output to files always uses the maximum output width, 132.

3.8. Sequential File I/O statements

The OPEN and CLOSE statements, and special specifications on the INPUT, PRINT, and LINE INPUT statements, as well as the function EOF permit BASIC programs to read and write system standard sequential files.

3.8.1. OPEN

```
OPEN <mode>,<number>,<file name>
```

The OPEN statement associates a file number with a file name and defines the mode which will be used to access the file. The <mode> expression must be a string, the first character of which is examined to determine the mode. If "I", the file is opened for input (to be read), and if "O", the file is opened for output (to be written). Under the Network Operating System, BASIC will automatically create a file opened with a mode of "O" if the file does not already exist. Under the Disc Executive, all files must be created from the system console before use. <number> is a numeric expression which must be greater than or equal to 0 and less than or equal to 15. This number is used to refer to the file once it has been OPENed. The <file name> is a string expression equal to the name of the file to be read or written. This may be a fully general file name acceptable to the operating system. The following two statements open file "DATA" for input as file number 1 and open file "ANSWER" for output

as unit 3.

```
820 OPEN "I",1,"DATA"
830 OPEN "O",3,"ANSWER"
```

The file number may be prefixed by a number sign (#) to be consistent with the use of file numbers in other statements.

3.8.2. INPUT

```
INPUT #<number>,<variable>,...
```

If the INPUT statement begins with a number sign, the expression following it will be used as the file number from which the input will be read. The file number used must have been previously opened in input mode by an OPEN statement. The format of data read from the file is identical to that accepted by the interactive form of the INPUT statement. The INPUT statement will always read at least one record from the file, and as many additional records as are required to satisfy the variables used on the INPUT statement. The following statement reads two numbers and a string from file number 1.

```
550 INPUT #1,I,J,V5$
```

If the end of the file is encountered, variables not yet read in will be left unchanged and the end of file flag will be set. This flag can be tested by the EOF function.

3.8.3. LINE INPUT

```
LINE INPUT #<number>,<string variable>
```

The LINE INPUT statement, when used with a file number specification, will read the next record of the file and store it in the named <string variable>. Only one <string variable> may be specified. If the end of the file is reached, the end of file flag (tested by the EOF function) will be set and the <string variable> will be unchanged.

3.8.4. EOF(<i>) - Function

The EOF function takes a file number (defined by a previous OPEN statement) as its argument, and returns 0 if the file is not at end of file and -1 if the end of file has been reached. This allows an IF statement to easily test whether an end of file was encountered in the last input from a file. The following program fragment reads in a file and stores it in an array until the end of file is reached.

```
400 LINE INPUT #5,S$(I)
410 I=I+1
420 IF NOT EOF(5) THEN 400
```

3.8.5. PRINT

```
PRINT #<number>,<expression>,...
```

If the PRINT statement begins with a number sign the expression following is taken to be the file number in which the output is to be written. The file number must have been previously opened in output mode with an OPEN statement. All the formatting options discussed for the PRINT statement in the section on Interactive I/O apply to the file PRINT statement.

3.8.6. PRINT USING

```
PRINT USING #<number>,<format>,<expression>,...
```

If the PRINT USING statement begins with a number sign the expression following is taken to be the file number in which the output is to be written. The file number must have been opened for output by a previous OPEN statement. The <format> and <expression>s are interpreted exactly like a PRINT USING directed to the system console.

3.8.7. CLOSE

CLOSE <number>,<number>,...

The CLOSE statement closes the files identified by the expressions listed on the CLOSE statement. For an input file, the file is simply detatched from the file number. For an output file, the last block is written to the file and the end of file sentinel is written into the file. Once a file has been closed, the file number may be reused for another file, and the file itself may be accessed again by issuing another OPEN with its name. Note that programs which write a file and then read it back MUST issue a CLOSE on the file before OPENing it for input. Failure to do this can result in reading past the end of valid data in the file and encountering garbage. For consistency, the file number expressions in the CLOSE statement may be prefixed by number signs (#). The following statement closes two files.

```
970 CLOSE 1,#3
```

3.9. Direct (Random) File I/O statements

Extended Commercial and Transaction BASIC allow use of Direct access (random access) files. These files are implemented in a manner largely compatible with sequential files, so large changes are not required in a program when converting back and forth between sequential and Direct file I/O.

Sequential files are stored with variable length records, and cannot be accessed except sequentially. Direct files are stored in fixed length records (the length is specified on the OPEN statement, see below). Associated with each Direct file is a read/write pointer, which contains the number of the record to be read or written next. When a file is OPENed, the pointer is set to record 0, which is the first record in the file. When a record is read by an INPUT # or LINE INPUT # statement, or written by a PRINT # or PRINT USING # statement, the record indicated by the pointer is read or written, and the pointer is incremented to point to the next record in the file. Hence, several INPUT or PRINT statements in a row will read or write consecutive records just as for a sequential file. The major difference between Sequential and Direct files is that INPUT and PRINT operations may be intermixed for Direct files, and that the read/write pointer may be manipulated by the SEEK statement described below.

3.9.1. OPEN (for Direct files)

OPEN <mode>,<number>,<file name>,<record length>

The OPEN statement for Direct files operates like the OPEN statement for Sequential files described in the section on Sequential I/O above. The <mode> must be "D" to indicate that the file being opened is to be a Direct file. The <number> is the file number to be used to access the file, and the <file name> is the string expression for the file to be opened. The Direct OPEN will error if the file designated by <file name> does not already exist. A BASIC program may dynamically create a Direct file by performing a Sequential OPEN ("O" mode), closing the file, then re-OPENing the file in Direct mode (this technique will work only under the Network Operating System, as the Disc Executive does not allow file creation from within a program).

The <record length> is an expression which will be truncated to an integer value and taken as the length in characters of records to be transferred to and from the Direct file. The actual records written to the file will be one character longer than the <record length> specification, as a

carriage return character will be appended to each record. Inserting this character allows BASIC-generated Direct files to be examined by other system utility packages, but has no effect on the BASIC program itself. When using BASIC under the Network Operating System, the <record length> may be any desired value, but under the Disc Executive, <record length> must be 127, as the Disc Executive does not allow byte-addressable files.

3.9.2. SEEK

SEEK <number>,<expression>

The SEEK statement will set the read/write pointer for the file indicated by the <number> expression to the integer part of the second <expression>. The next INPUT or PRINT will read or write that record (unless another SEEK is done first, of course). For consistency, the file number expression in the SEEK statement may be preceded by a number sign (#). The following statement will position file 7 to record 200.

```
2915 SEEK 7,200
```

3.9.3. LOC(<i>) - Function

The LOC function takes a file number (defined by the OPEN statement) and returns the current value of the read/write pointer for that file. LOC is defined only for Direct files. LOC may be used either to save record numbers in variables for later access, or in conjunction with the SEEK statement for relative record access. For example:

```
1820 SEEK #5,LOC(5)-1
```

will back up file 5 to the previous record.

3.10. Interprogram transfer statements

The CHAIN and COM statements (available in Extended Commercial and Transaction BASIC) provide a facility which permits one program to transfer control to another BASIC program stored in a disc file. The CHAIN statement performs the actual transfer to the other program, while the COM statement permits the declaration of common variables which are passed to the new program when it receives control.

3.10.1. CHAIN

CHAIN [*]<file name>[,<start line>]

The <file name> is a string expression which is taken to be the name of the disc file containing the program to which control is to be given. <start line> is an integer expression specifying the line number of the first line of the program to be executed after loading the new program. If <start line> is not specified, the new program will be executed starting at the first line. If an asterisk appears before the <file name> expression, the contents of the file will be merged with the current program, following the normal rules for BASIC line number editing. (That is, the resulting program will be the same as if the lines in the designated file were typed in from the keyboard while the running program was in the work area.) If no asterisk appears, the running program will be replaced by the program in the named file. CHAIN without the asterisk ("full CHAIN") is normally used when one program wishes to transfer control to another, while CHAIN with the asterisk ("partial CHAIN") is used primarily by programs to load common subroutine packages or configuration-dependent code.

When a full CHAIN statement is executed, all variables not previously declared by a COM statement (see below) will be deleted. Files opened by the original program will remain open, so the program CHAINed to can continue to access them. A partial CHAIN does not affect program variables.

3.10.2. COM

COM <variable>[(<expression,...)],...

The COM statement declares one or more variables to be "common". Common variables behave like any other variables, but are not deleted when a CHAIN statement is used to give control to another program. Hence, common variables may be used to pass parameters to the program being CHAINed to. Variables to be in common must be declared by a COM statement before being used in the program. COM variables may be either simple or subscripted. If subscripted, the dimension bounds must be specified on the COM statement; a DIM statement is neither necessary nor allowed.

3.11. Debugging statements

The debugging statements are provided to ease the debugging of a BASIC program. Program debugging usually involves use of immediate execution of statements (any statement entered without a line number is immediately executed). Refer to the section on immediate execution in Chapter 4 of this manual for more information.

3.11.1. TRON

TRON

The TRON statement turns on the BASIC statement trace. This will print each line before executing it. This allows the path of program execution to be determined when trying to locate a problem. The TRON statement can be placed before the section of the program known to have problems, avoiding tracing sections known to work properly. The TRON statement can be used in immediate mode to turn on the trace before executing a program with RUN. This will cause the entire program to be traced (unless it executes a TROFF statement to turn off the trace). If the BASIC source is not available (if the program was compiled and only the complied code was saved), the trace will print only the line numbers of the statements executed.

3.11.2. TROFF

TROFF

The TROFF statement turns off the statement trace enabled by the TRON statement.

3.12. Machine-dependent statements

The machine-dependent statements permit a BASIC program direct access to the memory and peripheral hardware of the computer on which it is executing. These statements allow great power in BASIC programming, but should be used only as a last resort, as they tie the BASIC program to the specific hardware configuration on which it was developed.

3.12.1. INP(<i>) - Function

This function returns the result from reading input port <i>. See the function section of this manual for a more complete description of INP.

3.12.2. OUT

OUT <port>,<data>

The <data> and <port> numeric expressions are evaluated. If either is

outside the range -32768 to 32767 an overflow occurs. Both values are truncated to the range 0 to 255 by clearing the upper 8 bits, then <data> is sent to the output port selected by <port>.

3.12.3. PEEK(<i>) - Function

This function returns the memory byte at address <i>. See the function section of this manual for a more complete description of PEEK.

3.12.4. POKE

POKE <address>,<data>

The <address> and <data> expressions are evaluated as numeric expressions. If the <data> expression is outside the range 0 to 255 an overflow will occur. The <address> may be within the range -32768 to 65535. The negative numbers address the memory locations corresponding to their 16 bit unsigned two's complement representation. Note that POKE, like PEEK, works on bytes.

3.12.5. WAIT

WAIT <port>,<mask>[,<xor>]

The WAIT statement evaluates the <port>, <mask>, and <xor> numeric expressions. If <xor> is omitted, zero is used. WAIT then enters a loop which reads from I/O port <port>, exclusive or's the value with <xor>, and's with <mask>, and continues to loop until the result is nonzero. When the result is nonzero, execution resumes with the next statement. The WAIT statement may be used to wait for a status bit to change on an I/O port, and permits much faster response than a BASIC loop using INP to read the port and the logical functions to perform the testing.

4. Using BASIC

The earlier sections of this manual have discussed how to write BASIC programs. This chapter will discuss how to enter, execute, modify, and save a program; in other words, how to use BASIC. BASIC is invoked from the operating system by typing its name:

```
BASIC
```

The operating system will load BASIC, and BASIC will issue its command prompt, which is a right corner bracket (>). When this prompt appears, BASIC is in command mode and ready to accept either a statement or a command.

4.1. Entering statements

Statements may be entered at any time simply by typing a line number followed by the text for the statement. The statement will be inserted in the position in the program indicated by its line number. If a statement already exists with the same line number as the statement just entered, the new statement will replace the existing statement. By using the AUTO command (discussed below), the user can cause BASIC to generate line numbers for statements automatically. This can eliminate the time-consuming and error-prone task of manually typing line numbers when entering a program.

4.2. Deleting statements

If a line number is typed with nothing following it, the statement with that line number will be deleted. If no statement exists with that line number, nothing will occur.

4.3. Immediate execution of statements

If a BASIC statement is entered with no line number, it will be executed in "immediate mode", that is, right away. The statement will operate in the context of the most recently RUN program. If no program has been run, the statement will execute in an empty environment. Immediate execution allows BASIC to be used as a desk calculator. An example of such use follows:

```
A=14
B=2
PRINT A,B,A*B
14            2            28
```

Immediate execution is also a powerful tool in program debugging. When a program errors and BASIC returns to command mode, all the program variables are left around with their values at the time of the error. They can be dumped or changed by the user with immediate mode commands, then the program execution can be resumed by typing in a GO TO statement in immediate mode. Sections of a program can be tested by inserting a RETURN statement after them, setting up the variables used by that section in immediate mode, and performing a GOSUB to the section to be tested. The section will be executed, and the RETURN will cause a return to command mode. The user can then examine variables with the PRINT statement to see the results of the execution of the section of program. Only one statement may be typed on a line when using immediate mode execution.

4.4. Commands

BASIC commands control the loading, saving, execution, and listing of BASIC programs. These commands may be entered when the BASIC command prompt (>) appears. BASIC commands may not be used as statements in a program.

4.4.1. AUTO

AUTO <start>,<increment>

The AUTO command will cause BASIC to automatically generate line numbers for statements typed by the user. If <start> is specified, the numbers will begin with that number, and each number will increase by the specified <increment>. If <increment> is omitted, the <increment> used in the last AUTO statement will be used. If this is the first AUTO statement, an <increment> of 10 will be used. If <start> is omitted, the line number of the last statement added to the program will be used, plus the <increment> in effect at that time. Once the AUTO command is given the user will be prompted with the line number for each statement. The user should enter the text of the statement. Entering a null line (just a carriage return), or the occurrence of a syntax error will terminate AUTO mode. If the line number prompt is followed by an asterisk (*), a line with that number already exists in the program and will be deleted if the user enters a statement. If this is not what the user intends, AUTO mode should be terminated by entering a null line.

4.4.2. BYE

BYE

BASIC will exit to the operating system. Be sure to SAVE your program before entering BYE if you want to preserve any changes made to it since the last SAVE done on it.

4.4.3. COMPILE

COMPILE:<program name>

Any program in the work area is discarded. The specified <program name> is read in and compiled to executable code. The source program will not be loaded into memory. If <program name> is not specified, the user will be prompted for it by the message:

Old program name?

Once a program has been COMPILEd, it may be RUN but not LISTed or modified, as the BASIC source code is not present. COMPILE is used to load debugged programs for execution, as it makes much more memory available for execution-time storage, since the source program is not in memory. For programs which are under development, the OLD command (see below) should be used.

4.4.4. CONTINUE

CONTINUE

The CONTINUE command is used to resume execution of a program halted by a PAUSE statement, a Control C from the console, or an execution error. If the program was halted by PAUSE or a Control C keyin, execution will resume following the statement at which the halt occurred. If the program was stopped by a runtime error, the CONTINUE command will reexecute the statement which failed, in the assumption that the user has corrected the problem that caused the failure by modifying that statement or changing variables by immediate mode statements.

4.4.5. LIST

LIST <start>,<end>

Lines of the program will be listed. If <start> and <end> are both omitted, the entire program will be listed. If only <start> is specified, just that line will be printed, and if both <start> and <end> are

specified, lines with statement numbers within the inclusive range from <start> to <end> will be listed.

4.4.6. NEW

NEW:<program name>

Any program in the work area is discarded, and a new program is defined with the specified <program name>. The <program name> should be the name of the file in which you intend to store the program. This may be any valid operating system file name. If the colon and <program name> is omitted, the user will be prompted for the program name with the line:

New program name?

If you have been modifying a program and wish the changes applied, be sure to do a SAVE before loading a new program with NEW. If you don't, the new program will overlay the old one in the work area and your changes will be lost without a trace.

4.4.7. OLD

OLD:<program name>

Any program in the work area is discarded, and a program is read in from the file <program name>. If <program name> is omitted, the user will be prompted for the program name by the message:

Old program name?

Any program previously in the work area will be overlayed by the OLD, so be sure to SAVE it if you wish changes made to it to be preserved. As the program is read in, it will be compiled and syntax checked, so if errors are detected in the OLD program, the error messages will be listed as the file is read. Any line in the OLD file which lacks a line number will be printed and discarded.

4.4.8. SAVE

SAVE

The program in the work area will be written to the file name specified as <program name> on the last OLD, NEW, or RENAME command. If none of these commands has been issued, the message:

Rename your program before saving it!

will be issued and the command will be ignored. The output file used by SAVE will be automatically created when BASIC is run under the Network Operating System. Under the Disc Executive, this file must have been previously created from the system console.

4.4.9. SAVEX

SAVEX:<file name>

The executable object code for the current program in the work area will be written to the file designated by <file name>. If <file name> is not specified, the user will be prompted for the file name by the query:

Object program name?

The executable object code is sufficient to execute the program, but does not contain the source code which is saved by the SAVE command. Files written by the SAVEX command may be loaded using the OLD or COMPILE commands (which are equivalent, since no source code exists in SAVEX files), or in properly-configured systems may be executed simply by typing

their name as a system command (the operating system will automatically load the BASIC execution monitor and run the program). The SAVEX command may be used for several reasons. First, programs saved with SAVEX are normally much more compact than the source code, so more programs and larger programs may be stored in the same amount of space. Second, programs saved by SAVEX cannot be listed or modified by the user (since no source code is saved), preventing theft or unauthorised changes in software provided to others. Third, since SAVEX-generated programs can be called directly from the console, BASIC programs can be called like any other executable programs without the user being required to learn how to invoke BASIC and OLD the desired program. The output file used by SAVEX will be automatically created when BASIC is run under the Network Operating System. Under the Disc Executive, the file must have been previously created from the system console.

4.4.10. SCRATCH

SCRATCH

All lines stored in the work area will be erased, but the current program name will be retained. This command is used when totally rewriting a program.

4.4.11. RENAME

RENAME:<program name>

The program in the work area is given the specified <program name>. A subsequent SAVE will store the program in the file by that name. If the colon and <program name> are omitted, the user will be prompted for the program name with the message:

New program name?

4.4.12. RUN

RUN

The program in the work area will be executed, starting with the lowest numbered statement in the work area.

4.5. Stopping a program

An executing BASIC program may be stopped by pressing the Control C key. This will cause the message:

Break in <line number>.

to be typed, and control returned to the command handler. The BASIC command prompt ">" will reappear. At this time, the user may use immediate mode statements to inspect and alter program variables, and may resume execution of the program by executing a GO TO from immediate mode.