RBASIC Enhanced BASIC

CONTENTS

Page

1. Preface 2
2. Introduction A-1
3. Control Characters A-1
4. DATA or INFORMATION A-2
4.1 Floating-Point Numbers A-2
4.2 Integer Numbers A-3
4.3 Strings A-3
4.4 Variable-Names A-3
4.5 Dimensioning of Variables A-4
5. Mathematical Operators A-4
5.1 Logical Operators A-4
5.2 Relational Operators (RELOPs) A-5
5.3 Total Operator Precedence A-5
6. DISK-FILE INPUT/OUTPUT A-6
6.1 SEQUENTIAL FILES A-6
6.1.1 Sequential File Input A-6
6.1.2 Sequential File Output A-7
6.2 RANDOM FILES A-8
6.2.1 Record I/O A-8
6.2.2 Virtual Arrays A-9
6.2.3 Virtual Array Disk Storage A-10
6.2.4 Extending Virtual Arrays A-11
6.3.1 Channel-Number Designation A-11
7. Channel #0 .... A-12
COMMANDS and STATEMENTS B-1
RBASIC Line Editor C-1
Adapting to your System D-1
Table of Statements, Functions and Commands E-1
Error-Messages E-2
Registration and Bug Reports F-1

1. PREFACE

Micronics Research Corp has designed RBASIC Enhanced BASIC to be upwards compatible with XBASIC*. It has **several** features not found in XBASIC, the most powerful of which is a built-in line editor, which is described in detail elsewhere in this Manual.

We have written RBASIC in compact 6809 code, unlike 6809 XBASIC, which, by and large,is a partial conversion of the earlier 6800 version into 6809 code. The merits of XBASIC are well established; it has from the very beginning served the 6809 community well as one of the fastest BASICs around. For several years, however, Micronics Research has been deeply involved in patching the original XBASIC - sometimes in response to correspondents' requests - either to correct discovered bugs or to add various enhancements. With time and experience, it eventually became apparent that continual patching was not the real answer, so we undertook the task of writing a new BASIC from the ground up, with two major considerations in mind :

a. RBASIC **must** retain compatibility with XBASIC, and
b. it should take up no more memory - preferably less.

As more and more features were added, these two restrictions made it difficult at times to achieve our objective, but the present version of RBASIC meets our criteria - with ample room to spare for future enhancements.

Depending on the type of program being RUN, RBASIC executes up to 16% faster than XBASIC, which may necessitate a corresponding adjustment to any delay-loops forming part of existing XBASIC programs.

Throughout this Manual all features of RBASIC not found in TSC's XBASIC are flagged by a '**' to the right of the feature's description. Simple enhancements to XBASIC features are similarly flagged with a '*'.

We welcome your comments or criticisms, including suggestions for enhancement, and especially reports of any bugs found.

* XBASIC Extended BASIC by Technical Systems Consultants, Inc, of Chapel Hill, North Carolina

2. INTRODUCTION

This Manual is not a tutorial in BASIC programming, but rather a reference describing in some detail the various features of RBASIC. To this end, apart from a short initial discussion, all commands and statements have been arranged in alphabetical order, instead of in functional groups. There are only two exceptions to this rule. One is the separate section on the EDIT command and its various control functions, as EDIT is really a program within a program, worthy of treatment in its own right. The other is the section dealing with Disk-File I/O, which is sufficiently complex to merit special treatment.

COMMANDS are instructions which are executable only in Immediate Mode, entered in response to the RBASIC prompt (for example, LOAD, RUN or LIST). They are non-executable if they form part of a program-line.

STATEMENTS, on the other hand, are executable either in Immediate Mode or may be entered in Program Mode as part of a program-line to be executed later when the program is actually RUN.

Instruction-lines beginning with a decimal-number (maximum 32767) are regarded by RBASIC as program-lines to be stored in memory for later execution. This form of entry known as PROGRAM-MODE. In IMMEDIATE-MODE, lines do not begin with a number and are "parsed" and executed immediately on detection of a CR entry, which indicates end-of-line. Parsing means that the line is examined for correct structure according to the rules of RBASIC.

In Program-Mode, lines may be entered in any order desired, as RBASIC automatically links them into correct numerical sequence. It is recommended, as good programming practice, however, that you number lines in increments of 10 to allow for the insertion of additional lines as your program develops. An existing line may be deleted by simply entering the line-number followed by CR, or may be over-written by entering the line-number followed by the new instruction-line, ending with a CR.

Lines may consist of one or more statements, multiple statements being separated by a colon (:) or a backslash (\). For example:

```
10  INPUT "What is your name",N$: PRINT "Hello there, ";N$
```

3. CONTROL CHARACTERS.

RBASIC uses five control characters in common with XBASIC. These are:

a. **^C** (Control-C), which causes the program to BREAK after completion of the currently-executing statement. If used while the program is waiting for input the program simply BREAKs. In both cases control is passed to the operator, who may then display, or even change, variable values (during a debugging session, for example), or list part of the program. Provided you make no changes to the program itself while halted in this way, the program will resume operation if you enter CONT (for CONTINUE). In this case, the program will resume at the **beginning** of the program-line at which it was HALTed. See BREAK.

b. **^H** (Backspace) which backspaces the cursor and erases the last-entered character. If entered at the start of a line, it issues a CR/LF.

c. **^M** (Carriage-Return) which causes a combined CR/LF to be issued to position the cursor at the start of a new line.

d. **^X** (Erase Line) which cancels the line currently being input from the keyboard and displays a ^X at the end of the line, followed by a CR/LF.

e. **^[** (ESC), which HALTs a program-listing as it scrolls on the display, or resumes listing if it has been so halted. While halted by an ESC, the listing may be aborted by hitting CR. This is the 'ESC-RETURN sequence'.

In addition there are several other (customisable) control characters implemented in the Line-Editor, for functions such as DELETE CHAR, INSERT CHAR, SPLIT or MELD, which are fully described in the section dealing with the Line-Editor. See also 'Adapting to Your System'.

4. DATA or INFORMATION

Information, or data values, may be represented as CONSTANTS or VARIABLES, each of which occurs as one of three types, FLOATING-POINT, INTEGER or STRING. An item of data whose value remains unchanged throughout the program (such as PI) is obviously a CONSTANT, while one whose value changes as the program progresses is a VARIABLE. However, RBASIC classifies them all as Variables, a CONSTANT being regarded as a 'fixed variable' and a VARIABLE as a 'variable variable'.

4.1 Floating-Point Numbers

Floating-point numbers are real numbers, i.e. those which normally include a decimal-point in their representation. RBASIC initialises itself to a display of 13 digits only, although this may be extended to 17 digits by the DIGITS statement. Internally, RBASIC computes to maximum precision within a dynamic range of approximately +/- 10^38, each number requiring 8 bytes of storage. You can enter numbers either directly as decimal numbers or in Scientific Notation of the form x.xxxEy, where the exponent 'y' may be either positive or negative. Numbers to be displayed in decimal are automatically converted to Scientific Notation if they are larger in magnitude than the number of digits set by the DIGITS statement or smaller than 1E-6. Some examples of valid Floating-Point numbers follow :

```
123.456              -1.234E5  or -1.234e+5
1.2345678            1E-3
.123456              1.        or 1.0
```

4.2 Integer Numbers

Integers are whole numbers in the range -32768 to +32767. Internally they occupy 2 bytes of storage. Examples are shown:

```
123                   12345
-123                  1
```

4.3 Strings

Whereas Floating-Point and Integer are restricted to numeric values, Strings can contain **any** displayable ASCII character, whether alpha or numeric. Examples of Strings are :

```
FRED                  Alpha123                    123
A1                    Hello there!                I'm O.K.
```

4.4 Variable Names

To keep track of the values of variables during program-flow, each variable must be assigned a Name. All names **must** begin with a letter of the alphabet, but may consist of up to two characters, the second of which may be either a letter or a single digit. Further, to distinguish between the 3 types of Variable, a final character may be added to the actual Name. Floating-point does not require a final character, but Integer-Names **must** end with a '%' sign, and String-Names with a'$' sign. Certain combinations of two letters are, however, not allowed, as they are reserved for use by RBASIC, such as AS, FN, IF, ON, OR, PI and TO. Some valid names are:

Floating-Point	A	I	AB	I5	but not	5	7A
Integer	A%	I%	AB%	I5%	but not	5%	7A%
String	A$	I$	AB$	I5$	but not	5$	7A$

The same alphanumeric combinations may be used within the same program for each of the three variable-types, as the final character (or absence thereof) serves to identify them uniquely. Thus A, A% and A$ may all be used within the same program without conflict.

4.5 Dimensioning of Variables

You may dimension any Variable with the DIM statement into a singly-dimensioned array (vector) or a doubly-dimensioned array (matrix). A vector is dimensioned by following the Variable-Name with a single integer enclosed in parentheses, while a 2-dimension array is dimensioned with two integers, separated by a comma. All dimensioned variables **must** be specified in a DIM statement before use. See 'DIM' for full details.

NOTE : A limitation is that the same Variable-Name cannot be used for both a 1-dimension and 2-dimension array.

5. Mathematical Operators

RBASIC has 5 mathematical operators, namely Addition (+), Subtraction (-), Multiplication (*), Division (/) and Exponentiation (^ or **), which are assigned the following precedence (highest to lowest) :

1.	()	Expressions in parentheses
2.	-	Unary Minus
3.	^	Exponentiation
4.	* /	Multiplication and Division
5.	+ -	Addition and Subtraction

When exponentiation is to be performed, the base is first converted to Floating-Point, then one of two operations takes place. If the power is also Floating-Point the result is calculated using LOG routines, but if the power is an integer a routine is called which essentially multiplies the base by itself the required number of times to produce a more precise result.

In addition, the residue of an Integer-Divide operation may be obtained. See MOD for full details. **

5.1 Logical Operators

RBASIC has four logical operators, namely AND, OR (inclusive OR), XOR (exclusive OR) and NOT, which are assigned the following relative priorities :

1.	NOT	Perform a Complement operation
2.	AND	Perform a logical AND operation
3.	OR XOR	Perform an Inclusive or Exclusive OR operation

These operators may carry out their functions on two numbers, in which case they perform the required operation in a bitwise fashion. Thus, given the following :

```
     A = 11 (1011)           B = 5 (0101)
then :

   NOT A  =  4 (0100)     NOT B   = 10 (1010)    A AND B = 1 (0001)
   A OR B = 15 (1111)     A XOR B = 14 (1110)
```

You may also use these logical operators to couple the truth-values of conditional statements, producing a result of '0' if the complete evaluation is FALSE, and a non-zero value if TRUE. See the individual definitions for full details.

5.2 Realational Operators (RELOPs)

RBASIC has six such operators, for use in conditional clauses to compare the relationship between two variables or constants for evaluation as TRUE or FALSE. RELOPs have equal mathematical precedence, and are set out below :

=	A = B	A equals B
<>	A <> B	A is not equal to B
<	A < B	A is less than B
<=	A <= B	A is less than or equal to B
>	A > B	A is greater than B
>=	A >= B	A is greater than or equal to B

If they are used to compare two numbers, the comparison will be done on the basis of their numerical value, but if used to compare two strings the ASCII values of the characters in the strings are used. Thus BOB is 'less than' BOY, but Bob is 'greater than' BOY because lower-case letters have a higher ASCII value. Strings, to be compared by RBASIC, must be of equal length, so if one is actually shorter than the other, RBASIC will pad it in memory with the required number of trailing SPACEs in order to equalise them.

5.3 Total Operator Precedence

The combined precedence of all operators is shown below, from the highest at the top to the lowest at the bottom. Operators appearing on the same line have equal precedence, and are evaluated from left to right as they occur in an expression.

1.	()		Expressions in parentheses
2.	^	**	Exponentiation
3.	-		Unary minus
4.	*	/	Multiplication and Division
5.	+	-	Addition and Subtraction
6.	RELOPs		Relational Operators
7.	NOT		The logical NOT operator
8.	AND		The logical AND operator
9.	OR	XOR	The logical OR and XOR operators

6. DISK-FILE INPUT/OUTPUT

In order to use **any** disk-file for the retrieval or storage of information, it must first be OPENed. There are two methods by which data may be stored on disk, Sequential I/O, the simpler method, and Random I/O, which is further subdivided into Record I/O and Virtual-Arrays. All such files default to the working-drive and to an extension of '.DAT'. No more than 12 I/O files may be OPEN at any one time (if memory-space allows), and, to keep track of them, they must be individually OPENed as 1, 2, 3 etc, though not necessarily in any particular numerical sequence.

A file is not actually OPENed at the time an OPEN statement is encountered. RBASIC merely prepares for the operation by assigning the specified Channel-Number to the named file, and reserving an I/O data-block for data transfer. RBASIC actually OPENs the file on the first attempt to write to, or read from,a particular channel.

6.1 SEQUENTIAL FILES

You may OPEN sequential files for either INPUT or OUTPUT, but not both. OUTPUT is indicated by OPENing a 'NEW' file, and INPUT by OPENing an 'OLD' file.

```
100  OPEN OLD "MYFILE" AS 1
100  I$ = "MYFILE": J% = 3: OPEN NEW I$ AS J%
```

Thus the first example above will prepare an already-existing MYFILE.DAT **for input only** on I/O Channel #1. The second example will prepare a new file "MYFILE.DAT" on I/O Channel #3 **for output only**. If a file of that name already exists on the disk **it will be deleted!**

6.1.1 Sequential File Input

```
100  OPEN OLD "MYFILE" AS 1
110  INPUT #1, I,J
```

In this example, MYFILE.DAT is opened on Channel #1, and two values read in and assigned to the variables I and J. Note that as the INPUT #1 statement requires 2 values separated by a comma, this is how the information should be stored in MYFILE (see Sequential File Output), though the values **may** be stored separately, separated by a CR. The important point to remember is that the data should be stored in ASCII (not binary) form, so that on INPUT the data appears to RBASIC as though it were coming from your terminal.

The information is read **sequentially** at each INPUT statement, much like the statement READ sequentially reads DATA lines in a program. To begin reading the same data a second time, i.e. to rewind the file, first CLOSE the designated Channel and then re-OPEN it.

You may use 'input' statements (other than 'INPUT'), such as INPUT LINE to input a line of maximum length 255, or INCH$(I%) to input a single character, where I% is the desired Channel-Number.

When a file is no longer needed, you should CLOSE it to free its Channel-Number for use by another file. Although each file **must** be OPENed individually, several may optionally be CLOSEd at one time with one CLOSE statement. For example :

```
500  CLOSE 1
500  CLOSE 1,5,8
```

6.1.2 Sequential File Output

```
100  OPEN NEW "MYFILE" AS 8
110  PRINT #8, "Hello there!"
120  PRINT #8, "How are you?"
130  CLOSE 8
```

The statement PRINT, followed by the desired Channel-Number, stores data in a sequential file. The above example stores two successive lines of data in the file MYFILE.DAT, a subsequent LISTing of this file from FLEX displaying

```
Hello there!
How are you?
```

exactly as it would have appeared on the terminal if the Channel # had not diverted the information to the disk-file instead. On the other hand, if Lines 110 and 120 had been combined so

```
110 PRINT #8 "Hello there!","How are you?"
```

a subsequent LISTing would produce instead

```
Hello there!    How are you?
```

The two strings would be combined into one in MYFILE.DAT, the ',' separator being converted to the requisite number of SPACEs to cause a TAB to Column 16 in the normal way.

If it's desired to input these two strings **as individual strings** on a subsequent 'INPUT #1' statement, such as

```
200  INPUT #1, A$,B$
```

then the ',' separator **must** be preserved, and not converted to SPACEs, by either of the two following forms, where the comma is forced.

```
        110  PRINT #8, "Hello there!,How are you?"    or better

        110  A$="Hello there!": B$="How are you?": PRINT #8, A$;",";B$
```

Similarly with numeric output to a file. The forcing-comma should be used to couple two different values, as in

```
        110  PRINT #8, A%;",";B%
```

6.2 RANDOM FILES

Random Files may be created as Record I/O or as Virtual Arrays.

```
        100  OPEN NEW "MYFILE" AS 1
        100  OPEN OLD "YOURFILE" AS 3
        100  OPEN "JOESFILE" AS 5
```

Unlike sequential files, the OPEN statement opens a file for both input and output, and occurs in three forms, as in the examples above. Random files may be read/written either sequentially or randomly.

Example 1 OPENs a new file MYFILE.DAT as Channel #1 on the working-drive. If a file of that name already exists **it will first be deleted!**

Example 2 instructs RBASIC to OPEN an already-existing YOURFILE.DAT as Channel #3 for input/output, while Example 3 (without either OLD or NEW) tells RBASIC to first look for JOESFILE.DAT, and, if found, to OPEN it for I/O as Channel #5. If no such file can be found, a new file with that name will be created and OPENed.

Just as with sequential files, you should CLOSE random files when they are no longer needed.

6.2.1 Record I/O

```
        100  OPEN "MYFILE" AS 3
        110  GET #3, RECORD 11
        120  PUT #3
```

Record I/O uses the GET and PUT statements to Input or Output data, and accesses the data in blocks of 252 bytes, which is the standard Record-Size. When first OPENed, a **new** Record I/O file will contain only one record, so if you need to create 30 records, a line such as

```
        105  PUT #3, RECORD 30
```

should be inserted at that time.

In the above example, MYFILE.DAT is OPENed for I/O as Channel #3, then Line 110 reads into that Channel's I/O buffer the 252-byte block of data constituting Record #11 and bumps the Record-Pointer to the next Record, i.e. Record #12. Line 120 then copies the I/O Buffer into Record #12. If, after appropriate data-manipulation, you wished to store back into Record #11, Line 120 should read

```
120  PUT #3, RECORD 11
```

after which the Record-Pointer would again be bumped to Record #12.

Attempting to access a non-existent Record-Number will produce an error-message, but you may extend a file by PUTting to a Record-Number higher than the highest-numbered Record currently in that file, which forces RBASIC to extend the file appropriately, plus a few extra sectors in anticipation of future extensions.

See CVT, GET, PUT and FIELD for more details.

6.2.2 Virtual Arrays

A virtual array is an array that resides on disk rather than in memory, but which can be referenced and manipulated similarly to a standard array. By this means arrays may be used which would otherwise be too large to fit in memory, and has the additional advantage that all data is preserved in a disk-file for use at some later date.

Virtual arrays, too, need to be DIMensioned and must also be associated with a specific I/O Channel before they can be used. The following

```
10  DIM #1, I(20,30)
20  DIM #2, J%(15)
30  I(20,30) = 0: J%(15) = 0
```

defines in Line 10 a Floating-Point matrix I of dimension 21 by 31 on Channel #1, while Line 20 defines an Integer array J% of dimension 16 on Channel #2.

The DIM statement does not, of itself, establish the size of the Virtual Array file, but merely its structure, and opens only one sector on disk for a new file being created. The file is extended, if necessary, when it's actually OPENed by making an assignment to the highest desired element, as in Line 30 above.

String arrays must be dimensioned differently, however, as each string in the array **must** be of the same length, with a maximum of 252 characters - unlike standard string arrays where the string may be of any length, and even change during program execution. Strings stored in a virtual array will be LSET. That is, they will be padded on the right with SPACEs if shorter than the defined length, or truncated to fit if longer. They are dimensioned thus:

```
10  DIM #5, I$(50) = 31
20  DIM #6, J$(20)
```

Line 10 assigns to Channel #5 an array I$ of 51 items of length 31 and Line 20 assigns to Channel #6 an array J$ of 21 items **of default length 18.** Strings **must** be contained within a single sector of 252 bytes, and so Line 10 assigns 8 strings to a sector, for a total of 8 * 31, or 248 bytes, thereby wasting 4 bytes per sector. On the other hand, Line 20 assigns 14 strings to a sector, for a total of 14 * 18, or 252 bytes, with no wastage at all. This, of course, implies that some thought should be given to the string-length assignment in order to maximise disk storage-space.

6.2.3 Virtual Array Disk-Storage

A virtual array file does not carry any information about its DIMensioning, the various data items being stored one after the other within each sector, together with any "wasted" bytes. 2-dimensional arrays are stored row-wise - that is, Row 0 is stored first, followed by Row 1, and so on. So if you required a print-out of a complete array it would be much faster to access the array row-by-row rather than column-by-column. In the latter case, the program would jump around all over the stored array to locate the various items, whereas in the first case, a simple sequential read-out would do the trick. Thus to obtain a printout of the items in virtual-array I$, you would use the following :

```
100  OPEN "MYFILE" AS 1          100  OPEN "MYFILE" AS 1
110  DIM #1, I$(10,10)           110  DIM #1, I$(120)
120  FOR I% = 0 TO 10            120  FOR I% = 0 TO 120
130  FOR J% = 0 TO 10            130    PRINT I$(I%)
140    PRINT I$(I%,J%)           140  NEXT I%: CLOSE 1
150  NEXT J%: NEXT I%: CLOSE 1
```

In the program on the left, Line 140 defines that the row index will advance the slower (because I% is specified before J%), so that data is read **exactly** as it occurs on the disk. If, however, I% and J% were swapped in this Line, the program would take considerably longer to execute. On the other hand, as was mentioned earlier, the data items **on disk** are not tied to any particular DIMensioning, so the right-hand program achieves exactly the same result, as MYFILE contains 121 items of data (11 * 11 = 121) according to the original assignment.

When a CLOSE statement is encountered in a program, the virtual array associated with that Channel is also closed, and will therefore need to be DIMensioned once more if that file is re-OPENed at some later stage.

6.2.4 Extending Virtual Array Files

Virtual array files, when first created, are assigned sufficient sectors to accommodate the highest numbered element originally assigned to the array, plus a few extras, and must be extended to accommodate additional data items. The process is quite simple, involving nothing more than making a dummy assignment to an array-location beyond the end of the current file.

```
100  OPEN "MYFILE" AS 1
110  DIM #1, I$(100)
120  I$(100) = "Hello!"
130  PRINT I$(100): CLOSE 1
```

Assuming that MYFILE.DAT currently holds only 20 data items, Line 120 forces an extension to 100 items. If it were not for Line 120, Line 130 would respond with an error-message, because array item #100 would be non-existent. However, Line 120 forces RBASIC to extend MYFILE to accommodate 100 items, plus a few extra sectors to avoid the need for future extensions, which can take some time to accomplish. Line 130 can now read item #100 from disk and print out the word "Hello!". Of course, Line 120 could equally as well assign the NUL-string "" to I$(100), which would leave us with a virtual array composed of the original file, plus an extension to 100 elements composed of nothing but NUL-strings.

6.3.3.1 Channel-Number Designation

Throughout the whole of this Section on Disk I/O, Channel-Numbers have been specified as integers, e.g., INPUT #3, PRINT #8, GET #3, DIM #1, etc., to simplify the explanation of the various forms of Input-Output.

The integers following the '#' sign may, however, be replaced by any legitimate expression **provided the result of the evaluation does not fall outside the range of allowable Channel-Numbers,** that is, 1 - 12 for Disk I/O, or the special Channel-0 discussed in the next Section. Fractional evaluations will automatically be integerised. The following examples are all valid forms :

```
10 N%=3: PRINT #N%, A$          10 N%=3: PRINT #N%+3, A$

10 PRINT #2*PI, A$              10 N%=2: X%=3: PRINT #N%*X%, A$
```

The two examples in the first row would output A$ to Channels 3 and 5 respectively, while those in the second row would both output A$ to Channel 6.

7. CHANNEL #0

Recall that **disk-files** are OPENed for Input or Output on Channels 1 - 12 only. Channel #0 has a special significance in RBASIC, in that it allows a request for input without the accompanying '?' prompt when used with the INPUT statement. See 'INPUT' for full details.

When coupled with the PRINT statement, however, Channel #0 allows output to be directed to some device other than your terminal, for example a Printer. Thus

```
100  OPEN "0.PRINT" AS 0
110  PRINT #0, "Hello there, Printer"
120  PRINT "Hello, Terminal"
130  CLOSE 0
140  PRINT #0, "Hello again, Terminal"
```

causes RBASIC to OPEN and read in 0.PRINT.SYS (.SYS being the default extension), then Line 110 outputs the message "Hello there, Printer" to the printer instead of to your terminal. Line 120 (in the absence of '#0') directs its message to your display-screen in the normal way, while Line 130 CLOSEs off 'PRINT.SYS'. Line 140, even though it still has '#0' appended to the PRINT statement, displays its message on your screen, because Channel #0 is no longer OPEN.

The following example demonstrates how, by using Channel #0, you may select where a message is to be directed.

```
100 INPUT "Output to Terminal, Modem or Printer (T, M or P) ",Q$
110 IF Q$ = "M" THEN OPEN "0.MODEM" AS 0
120 IF Q$ = "P" THEN OPEN "0.PRINT" AS 0
130 PRINT #0, "Hello, whoever you are!"
140 CLOSE 0
```

Depending on your response at Line 100, output will be directed appropriately. For example, if 'T' were entered, the program falls through Lines 110 and 120, and displays the message on your terminal, whereas "P" OPENs and reads-in PRINT.SYS on Channel #0, causing the message to be directed to your Printer instead. Similarly, a response of "M" sends the message out via your Modem.

COMMANDS AND STATEMENTS

```
+         +CAT
          +RENUM
          100  A$ = B$ + "X" + C$
```

The '+' **command** in the first two examples transfers the remainder of the command-line to FLEX for execution. You should ensure that the command(s) to be executed reside entirely within the Utility Command Area of FLEX. '+' is normally used to invoke the RENUM command for renumbering the lines in an RBASIC program.

Example 3 shows how the '+' **statement** may be used as a 'concatenate' operator to join B$ and C$, with an 'X' between, into a new string A$.

See also RENUM

!

```
          10 ! "Your name ",N$                                          **
          10 INPUT "Your name ",N$
```

The '!' statement is equivalent to typing in the word 'INPUT', and is internally converted to 'INPUT' by RBASIC. Thus the two example lines above are equivalent.

?

```
          10 ? "Hello"
          10 PRINT "Hello"
```

The '?' statement is equivalent to typing in the word 'PRINT', and is converted to 'PRINT' by RBASIC. The two example lines above are therefore equivalent.

ABS(X)

```
          100 PRINT ABS(-3.5)
          100 PRINT ABS(I)
```

The ABS statement removes the '-' sign (if one exists) from the argument enclosed in parens, thus converting it into a positive number of equal magnitude. Positive numbers are left unchanged. The first example returns a value of 3.5, while the second converts the value of 'I', whatever it may be, into an equivalent positive number.

AND

```
          100 PRINT 7 AND 5
          100 IF Q$="Y" AND R$="Y" THEN PRINT "OK"
```

The AND logic-operator performs a bit-wise AND operation on the Integer values of numbers, or on the truth-values of conditional clauses. Thus example one produces a result of '5' when ANDing 7 (111) with 5 (101) because only in the first and last bit-positions does a 1-bit exist in 7 AND in 5. The second example produces a truth-value of TRUE and prints 'OK' only if both Q$ AND R$ are equal to "Y", otherwise the test fails and the program falls through to the succeeding line.

APPEND APPEND "TEST" **

The APPEND command is very similar to LOAD, in that it LOADs the named File into the Source-Buffer. Instead of deleting the existing program before LOADing, however, it APPENDs the new program to the end of the current .BAS file in memory. Line-Numbers in the file to be APPENDed **must** be higher than the highest line-number of the current program. This command is very useful for the serious programmer who wishes to create a library of often-used routines, each with its own specific block of line-numbers, to be available for APPENDing (and possible RENUMBERing later) during new program development.

ARC 100 PRINT ARC SIN(X), ARCCOS(X), ARCTAN(X) **

The ARC statement causes the inverse of the named Trigonometric function to be performed, the result being returned in radians or degrees, depending on the setting of the DEGREES/RADIANS switch. Note that the use of a SPACE after ARC is optional.

See DEG, SIN, COS, TAN

AS See FIELD, OPEN

ASC(I$)
```
100 I$="Hello": PRINT ASC(I$)
100 PRINT ASC("Hello")
```

The ASC function is the inverse of CHR$, and returns the **decimal** ASCII value of the first character in the named String. If the String does not exist, or is the NUL-string, a value of 0 will be returned. In both the examples given, the value printed is 72, the ASCII value of 'H', the first character in "Hello".

ATN(I) 100 PRINT ATN(2.5)

The function ATN calculates the ARC-TANGENT of the argument enclosed in parens, displaying the result in radians or degrees, depending on the setting of the DEGREES/RADIANS switch. ATN is retained solely for compatibility with XBASIC, and would be more logically expressed as ARCTAN in RBASIC. **

See DEG, TAN

BREAK ^C

If you operate the BREAK key while an RBASIC program is actually RUNning, the program will be interrupted and break off with a message such as:

```
   BREAK AT LINE 100
```

Similarly if you BREAK while the program is temporarily waiting for operator-response to an INPUT type of statement. In either case, you may examine **or change** the values of desired variables, and the program then made to continue by entering CONT. The program will resume at **the start of the line** at which it was halted by BREAK.
See CONT

CHAIN

```
2000 CHAIN "CHESS2"
2000 CHAIN "CHESS.TWO" 100
2000 CHAIN "CHESS2.BAC" 300
```

Sometimes a program which is too large to fit in memory at one time can be conveniently split into two or more subsections, each of which is stored as a separate file on disk. The CHAIN statement allows these subsections to be called up in sequence, with a specified commencing Line-Number if desired.

If no extension is given, as in example one, RBASIC assumes that the file-type is the same as that of the calling program (that is, either . BAS or .BAC), and similarly for example two - where the extension is not specifically .BAS or .BAC. Example 3 is used when you wish to switch from one file-type to another.

Where no line-number is specified (Example 1) the CHAINed program begins execution at the lowest line-number, whereas Example 2 commences execution at Line 100 of the CHAINed program, and Example 3 at Line 300. No parameters can be passed between CHAINed files other than via disk-files, and any files to be shared by CHAINed files must be specifically OPENed in each section called, as RBASIC closes off **all** files before loading in the new subsection.

CHR$(I%)

```
PRINT CHR$(7)
PRINT CHR$(72)
```

The CHR$ function is the inverse of ASC, and returns a single ASCII character corresponding to the argument I%. Example 1 outputs the 'BELL' character and causes your terminal to 'beep', while the second prints the ASCII character 'H'.

CLEAR

```
CLEAR
```

The CLEAR command is automatically carried out each time RUN is executed, and initialises to zero all the program variables.

CLOSE

```
100 CLOSE 1
100 CLOSE 1,2,3
```

The CLOSE statement closes a file after it has been OPENed on a specified Channel, thus freeing up the Channel for use by another file. Example 1 above closes the file previously OPENed up on Channel 1, while Example 2 closes the files previously OPENed on Channels 1, 2 and 3.
See also OPEN, DISK-FILE INPUT/OUTPUT Pages 7 – 12

COMPILE COMPILE "CHESS"
COMPILE "2.CHESS.NEW"

The COMPILE command saves a file to disk in a compiled (compressed code) form, and defaults to a .BAC extension unless specified otherwise. Depending on the nature of the program, the compiled program is normally much shorter than the uncompiled form, and both loads and executes much faster. When COMPILing a program, you should also SAVE the original source with the SAVE command, as it is not possible to edit the COMPILEd form. Editing can only be carried out on the original source program, which should then be re-COMPILED. Unlike XBASIC, a COMPILEd program may be LISTed for informational purposes only. **

CONT CONT

The CONT command causes resumption of program operation when it has been interrupted either by a BREAK (^C) from your keyboard or by a STOP statement in a program-line. Provided no changes have been made to the program itself while halted, resumption occurs in the first case at the start of the line in which the BREAK was encountered, or at **the line immediately following** in the case of a STOP.

COS(I)

```
10 PRINT COS(2.3)
10 PRINT COS(I+PI/2)
```

COS calculates the trigonometric COSINE of the angle I, which is expressed in radians or degrees, depending on the setting of the DEG/RADIANS switch.

See ARC, DEG

CVT$(I%)
CVTF$(I)
CVT$%(I$)
CVT$F(I$)

The first two CVT statements convert numeric data to string data for storage in the I/O buffer, while the last two perform the reverse operation. They are quite different from VAL and STR$, which work with ASCII and must therefore carry out an ASCII-to-binary conversion, thus requiring a lot more time. The CVT functions, on the other hand, work directly with internal binary, and are therefore much faster.

The following example shows how to transfer the data in a FP numeric array to a 'record I/O' type of file:

```
   10  DIM A(30), A$(30)
    .   .    .       .
  100  OPEN "MYFILE" AS 1
  110  FOR I%=0 TO 30:  FIELD #1, 8*I% AS I$, 8 AS A$(I%): NEXT I%
  120  FOR I%=0 TO 30: LSET A$(I%) = CVTF$(A(I%)): NEXT I%
  130  PUT #1: CLOSE 1
```

Line 10 DIMensions the FP array A, which will be filled with data by the lines between 10 and 100, and also the string-array A$, in which the converted data will be stored. Line 100 OPENs up file "MYFILE" as Channel 1, ready to receive the data, while Line 110 partitions the I/O buffer into 31 blocks of 8-bytes. This is the maximum which can be stored here, as the maximum Record-size is 252 bytes. Then Line 120 converts each numeric item in the FP array 'A' into its corresponding string, which is LSET into the 8-bytes reserved for it in the I/O buffer. When all 31 items have been converted and stored, Line 130 transfers the contents of the I/O buffer to the disk-file "MYFILE" as one complete record.

The reverse operation, using CVT$F, is used to read back the data at some later time. Note that if the converted data has less than 8 bytes (2 if we were dealing with Integer arrays) the string will be padded on the right with NUL characters.

DATA

```
10 DATA 1.5,5,1.23E4
10 DATA 29,MAY,1986
10 DATA "  It's 29 May, 1986"
```

The DATA statement defines items of information which will be used by the program when it executes a READ statement. RBASIC maintains a DATA pointer, initialised to the first one encountered in the program, and updates this pointer to the next item each time a READ is executed. In the case of Example 1, successive READs pick off 1.5, then 5, then 1.23E4, while Example 2 successively picks off 29, then MAY and finally 1986. Example 3 demonstrates how to enclose SPACEs, quotes and commas within a string.

DATA statements are not allowed in multi-statement lines.

See also READ, RESTORE

DATE$

```
PRINT DATE$
100  I$ = DATE$: PRINT I$
```

DATE$ reads the current date in FLEX's date-registers, and returns it in the format 'DD-Mon-YY'.

DEF
```
100 DEF FNA(R) = PI*R*R
100 DEF FNB(X) = RND(0)*X+1
```

The DEF statement defines a **single-line** function which may occur repeatedly in a program, its name being formed of the letters 'FN' followed by any legal FP variable-name. The argument enclosed in parens has no significance to the program, its purpose being only to specify to the function-line the operation to be carried out on whatever argument is passed to it. The value returned by the function must also be Floating-Point.

Thus, even though the argument in example 1 (which returns the area of a circle of radius R) is specified as 'R', the function can be called by either of the program lines shown below, both of which would assign the same value to A, namely PI*3*3+5 :

```
200 A=FNA(3)+5
200 B=3: C=5: A=FNA(B)+C
```

The second of the first set of examples is useful for producing a random number in the range 1 to X. Random **integers** from 1 to 10 would be produced by a line of the form:

```
200  R%=FNB(10)
```

Note the use of R% to produce an integer from a FP function.

DEG
```
100  DEG0                                                 **
100  DEG 1                                                **
```

The DEG statement sets the Trigonometric functions to "Degrees" or "Radians" mode. RBASIC initialises itself to "Radians" when loaded into memory, and also re-initialises to "Radians" each time a file is LOADed - whether a .BAS or .BAC type. Therefore, if files are CHAINed and 'DEGREES' is to be assumed, you should re-specify 'DEG 1' in each CHAINing. Example 1 above sets RBASIC to "Radians", while Example 2 sets it to "Degrees". Note that a SPACE is optional after the DEG statement.

DIGITS
```
100  DIGITS 17
100  DIGITS 8,2
```

The DIGITS statement tells RBASIC how many digits to display or print, though full precision is maintained internally. The maximum is 17, though this is not recommended, as rounding errors can cause the 17th digit to be incorrect. The minimum number is 1.

Example 1 sets the DIGITS value to 17 (the maximum), whether to the right or left of the decimal-point, whereas Example 2 specifies a maximum of 8 digits to be displayed, rounded to a maximum of 2 places to the right of the decimal-point. Obviously the second digit **must** be less than or equal to the first digit specified.

A number outside the range encompassed by the DIGITS setting will be displayed in scientific notation.

DIM
```
10  DIM A(19),B%(29),C$(19,29)
```

The DIM statement defines the dimension of a subscripted variable. All such variable-arrays have a base-subscript of '0', so the example line would define an 'A' array of 20 variables named A(0), A(1), A(2) .... A(19), a B% array of 30 variables named B%(0), B%(1), B%(2) .... B%(29), and a C$ array of dimension 20 x 30, set up as below :

```
C$(0,0)   C$(0,1)   C$(0,2)   ....   C$(0,29)
C$(1,0)   C$(1,1)   C$(1,2)   ....   C$(1,29)
C$(2,0)   C$(2,1)   C$(2,2)   ....   C$(2,29)
   .         .         .                .
   .         .         .                .
C$(19,0)  C$(19,1)  C$(19,2)  ....   C$(19,29)
```

Where possible, numeric arrays should be restricted to integers, which use up only 2 bytes of memory per integer, compared to 8 bytes for each FP number.

Subscripted variables may use the same name as non-subscripted ones, but a singly-dimensioned variable may not have the same name as a doubly-dimensioned one. Thus it is possible to use the names A1% and A1%(5) in the same program, but this would exclude use of the name A1% for a 2-dimension array, and vice versa. All variable arrays **must** be DIMed before they can be used in RBASIC programs, and once DIMed they cannot be re-DIMed. There is no default dimension of 10 (or 10 x 10) as with some other BASICs.

See also VIRTUAL ARRAYS

DISK I/O See Special Section

DOS
```
DOS
100  IF A > B GOTO 10 ELSE DOS                                **
```

The DOS **statement** replaces XBASIC's FLEX **command**. It exits RBASIC and returns control to FLEX. Note that 'DOS' can now function within a program, as well as in Direct Mode.

DPEEK(I)
```
100  X = DPEEK(HEX("E010"))
100  A = HEX("E010"): X=DPEEK(A)
```

The DPEEK function is the opposite of DPOKE, and returns the contents of the double-byte stored at address I. The value returned will be a decimal number in the range 0 - 65535. Maximum value for the address I is 65535, or $FFFF.

See also PEEK, POKE, DPOKE

```
DPOKE    100  DPOKE HEX("E010"),1234
         100  DPOKE A,X
```

The DPOKE statement is the opposite of DPEEK, and stores a double-byte of data at a named decimal address. Data may be passed in this way to either a machine-language program or to an output-port. The maximum value for both the address and the data to be passed is 65535, or $FFFF.

See also POKE, PEEK and DPEEK

e 2.7182818284590452

Although 'e' is not specifically included in RBASIC, it may easily be implemented by setting a variable, say E, to its value, thus:

```
         10  E = EXP(1)
```

EDIT **

This command invokes the built-in Line-Editor. See the special section at the end of this Manual for full details of this feature.

ELSE See 'IF'

END 100 END

The END statement is optional in an RBASIC program, its purpose being to terminate program-execution and return to the RBASIC prompt. No BREAK message is output, and the program cannot be re-started with CONT.

ERL See ERR

ERR 100 IF ERR = 30 AND ERL = 50 THEN PRINT "Data Type Mismatch"

The variables ERR (Error-Number) and ERL (Error Line-Number) are updated each time an error is encountered in a program. ERR contains the error-number, and ERL the number of the line in which the error occurred. In the example given, the error-message will be displayed if error-number 30 occurs in Line 50.

See also ON ERROR, RENUM

ERROR-TRAPPING

The following program demonstrates how to use the 'ON ERROR GOTO' statement to trap various kinds of error :

```
  10 ON ERROR GOTO 100
  20 INPUT "Enter a number",I
  30 PRINT "The square-root of";I;"is";SQR(I): GOTO 20
 100 IF ERR = 30 THEN PRINT "Numbers only, please!": RESUME
 110 IF ERR = 34 THEN PRINT "OK. We'll try a new game!": RESUME 200
 120 ON ERROR GOTO
 200 Guessing-game program ...
```

If a non-numeric entry is made at Line 20, control is passed to Line 100 which checks to see if it's a Data-Type error, and if so causes the appropriate message to be displayed, then returns control to the start of Line 20. On the other hand, if a negative numeric entry had been made, the error-handling routine would fall through to Line 120, and return control to RBASIC to handle Error 107 (Imaginary Square Root).

A 'BREAK' (^C) response at Line 20 would be trapped by Line 110, which would display its message and then resume at Line 200, which represents the start of a number-guessing type of game. Of course, by trapping ^C in this way, it is not possible to exit the program cleanly in any way, as the program will always keep cycling through Line 110 and returning to the start of the guessing-game. One way to avoid this would be to re-write Line 110 so :

```
  110  IF ERR=34 AND ERL=20 THEN PRINT "OK ... etc
```

In this case, ^C would be trapped should it occur at Line 20, but the new line 110 would allow ^C to act as a means of exiting once program-flow transferred to the guessing-game.

EXEC
```
         100  EXEC, "CAT 0"
         100  EXEC "CAT 0: CAT 2"                                 **
```

The EXEC statement executes the FLEX utility-command(s) enclosed in quotes. You should ensure that the named utility resides entirely in the the FLEX utility command area, otherwise RBASIC itself or part of its program may be overwritten, with unpredictable results. Note that the use of a comma following 'EXEC' is optional in RBASIC, as in FLEX.

EXP(I) `10  J = EXP(3.5)`

This function calculates e^I, where 'e' is the base of 'natural' logarithms, approximately 2.718281828459045. The maximum allowable value of I is approximately 88.

FIELD `100  FIELD #1, 8 AS I$, 10 AS J$, 2 AS K$, ...`

Whenever the OPEN statement opens a Record I/O File, a 252-character I/O Buffer is created and allocated to that Channel for temporary storage and manipulation of the Record. The FIELD statement associates String-Names with various sections of this Buffer in accordance with the lengths specified.

Thus in the example above, the first 8 characters of the I/O Buffer are associated with I$, followed by 10 characters for J$ and 2 for K$, etc., to a maximum of 252 characters.

If a subsequent GET statement loads the Buffer with new data, the old is over-written with a new distribution of the data being FIELDed. You should keep in mind that the FIELD statement does not directly manipulate the data - it simply sets up the FIELD-sizes. If the I/O Buffer is currently filled with data and a new FIELD statement is executed, the FIELD boundaries are simply reallocated over the Buffer's contents, without changing the actual position of any of the characters stored.

If, while a String-Name is associated with the I/O Buffer, it is assigned a value via a LET (or implied LET) statement, that String-Variable is effectively removed from the FIELD definition. So, in the example

```
100  OPEN "MYFILE" AS 1
110  FIELD #1, 8 AS I$
120  I$ = "Hello"
```

line 120, by assigning the value "Hello" to I$, serves only to cancel the FIELD definition of Line 110. I$ may, however, be used in this way once all operations on MYFILE in Channel 1 have been completed and the file closed off. The correct way to modify the contents of the I/O Buffer is by use of the LSET or RSET statement.

If a record-size of 252 is too large, the I/O Buffer may be split up into sub-records, each of which **must** be of the same length. It may, for instance, be broken up into 8 sub-records of 31 characters each (for a total of 248 characters), leaving 4 characters undefined. The following example shows how to accomplish this :

```
100  DIM J$(7), K$(7): OPEN "MYFILE" AS 1
110  FOR I% = 1 TO 8
120    FIELD #1, (I% - 1) * 32 AS Z$, 24 AS J$(I%), 8 AS K$(I%)
130  NEXT I%
```

In Line 120, Z$ is a dummy variable used to format the I/O Buffer correctly. When I% is 1, Z$ is allocated a length of 0 characters, and so J$(1) becomes associated with the first 24 characters of the Buffer and K$(1) with the next 8. When I% is equal to 2, Z$ is allocated a length of 32 characters, so that J$(2) and K$(2) become tacked on to the end of K$(1), and so on.

See also LSET, RECORD, RSET

FOR

```
        100 FOR I% = 1 TO 10
        200    program instructions
        300  NEXT I%

        100  FOR I = 2 * X TO N - 3 STEP 2.5

        100  FOR I% = 5 TO -2 STEP -1
```

The FOR statement causes a counter to be set up and initialised, before executing the instructions between it and its corresponding 'NEXT'. When 'NEXT' is encountered the counter-value, or index, is increased (or decreased) by the 'STEP' value and compared with the defined end-limit. Provided the adjusted index does not break the end-limit, the cycle repeats until such time as the limit **is** broken, at which point program-control resumes at the statement following 'NEXT'. Where no 'STEP' size is specified a default-value of '1' is assumed.

All FOR-NEXT loops are executed at least once, even though the initial value is already beyond the defined limit!!

Thus in the 3-line Example 1 above, Counter-I% is initialised to 1, the following instructions executed, then 'NEXT I%' bumps the index to 2, and returns control to the 'FOR' statement. The cycle is repeated 10 times before I% exceeds the upper limit of 10, causing control to be transferred to the line following Line 300.

Example 2 demonstrates that any valid numeric expression (FP or Integer) may be used to set the initial value and upper-limit, and also the STEP-size, if so desired.

Example 3 shows how a reverse-count is established by setting the initialising-value higher than the limiting-value and using a negative STEP-size. In this instance the FOR-NEXT loop repeats until the index-value falls **below** that of the specified limit.

FOR-NEXT loops may be nested to a level limited only by the amount of memory available, with two further restrictions :

(i) The same index-variable may not be used in more than one level.

(ii) The corresponding 'NEXT' for any index-variable may not occur outside the nested loop. Thus the following program is invalid:

```
   100  FOR I% = 1 TO 10
   110    FOR J% = 1 TO 5
   120      program instructions
   130    NEXT I%
   140  NEXT J%
```

The statement 'NEXT J%' **must** occur before 'NEXT I%' to ensure that the j%-loop is truly wholly nested within the i%-loop.

See also NEXT

FRE(0)   PRINT FRE(0)

The function FRE(0) returns the number of free bytes of memory currently available. The argument '0', although not used, is necessary for the sake of compatibility with other functions.

```
GET       100  OPEN "MYFILE" AS 1
          110  GET #1, RECORD 5
           .    .   .    .    .
          250  GET #1
```

The GET statement accesses a given record in Record type I/O. Records are numbered from 1 to n, where n is the file-size in records. The example above causes Record #5 to be accessed and stored in the I/O Buffer for further manipulation. If no Record-Number is specified, as in the example Line 250, the next available record will be accessed (if there is one available).

See Special Section on Disk-File I/O

```
GOSUB    100  GOSUB 500
```

The GOSUB statement transfers program-control to the named Line-Number immediately following. Upon encountering a RETURN in the subroutine, control will be returned to the statement following the GOSUB which called it.

See also ON-GOSUB, RETURN

```
GOTO     100  GOTO 500
```

The GOTO statement causes program-control to be directly transferred to the named Line-Number. Unlike GOSUB, there is no RETURN involved. because of this any statements following GOTO on the same line **will never be executed.** See the section on IF-THEN-ELSE for a way around this restriction.

See also ON-GOTO

```
HEX(I$)   100  PRINT HEX("FFFF")
          100  A$ = "FFFF": PRINT HEX(A$)
```

The HEX function converts the character-string enclosed in parens into its equivalent decimal number. Both examples above return the value 65535, which is the decimal equivalent of $FFFF.

```
IF        10  IF A > 3 GOTO 100
          10  IF A < B GOSUB 100                                  **
```

The RELOP in each of the above examples is evaluated, and if TRUE, would, in example 1, cause Line 10 to GOTO line 100, or, in example 2, cause Line 10 to GOSUB Line 100. GOTO and GOSUB are the only two adjuncts of 'IF' which do not require a 'THEN' statement.

XBASIC, however, requires 'THEN' before a 'GOSUB' statement.

```
10  IF A > B THEN 100
20  IF A$ = "YES" THEN PRINT "OK"
30  IF A = 9 THEN A = 0: B = 15
40  IF A = 1 THEN IF B = 2 THEN RETURN
```

This form of 'IF', using 'THEN', will, in general, permit **any** valid statement to follow the 'THEN'. This statement, or statements, will execute if the conditional clause evaluates to TRUE. Note that in the first example, 'THEN' has an implied 'GOTO' following it, and causes the program to 'GOTO 100'. Example 4 demonstrates the use of nested 'IF's, though it would, of course, be simpler to say 'IF A=1 **AND** B=2 THEN RETURN'.

```
10  IF A <= B GOTO 100 ELSE PRINT "A > B": GOTO 100
```

In this example, Line 10 transfers to Line 100 if 'A<=B' were TRUE, but instead of falling through to the next line if it were FALSE, the instructions to the right of 'ELSE' would be executed. Only one valid instruction may be placed **between** 'IF' and 'THEN'. Thus the following is invalid:

```
10  IF A <= B THEN X = 2: GOTO 100 ELSE PRINT "OK": RETURN
```

Just as in the earlier examples, IF-THEN-ELSE statements may also be nested. Thus:

```
10  IF A = B THEN X = 1 ELSE IF A = C THEN X = 2 ELSE X = 3
```

INCH$(I%)
```
100 Q$ = INCH$(0)
100 PRINT "Enter a number (1 - 9) ";: Q$ = INCH$(0): PRINT
```

The INCH$ function inputs a single-character response from Channel I%, 0 being your terminal. No final CR is expected, a single character (which may be a CR itself) is all that is necessary. No input message may be used, and no prompting '? ' will be issued. Example 2 illustrates how to use the PRINT statement to display a message-string. Note the ';' immediately after the message-string to cause the cursor to hold at that point. Because a CR may not necessarily be the actual response, a final PRINT is sometimes necessary, otherwise the next message-string occurring in the program may overlay the one presently displayed.

See also INPUT, INPUT #I%, INPUT LINE, BREAK and CONT

INPUT
```
100  INPUT N$
100  INPUT N$,A%,B
100  INPUT 'Your name is',N$
100  INPUT "Your name",N$; "Your age",A%; "Thank you!"
```

The INPUT statement outputs the message-string enclosed in quotes (if there is one), appends a '? ' and then waits for an appropriate operator-response. Note that either single or double quotes may be used to enclose the message-string.

The final example is equivalent to the multiple lines :

```
100 INPUT "Your name",N$
101 INPUT "Your age",A%
102 PRINT "Thank you!"
```

If more than one response is expected (as in Example 2) each one should be separated from its predecessor by a comma and the final entry terminated with a CR. If you make too few responses when CR is entered, RBASIC prompts with another '? ' for further entries. If you make too many responses the extras are ignored.

A simple CR is not acceptable as a response - INPUT **must** have an actual response (which may be a complete line in response to a $-type request) followed by a CR.

See also INPUT #(I%), INPUT LINE, INCH$, BREAK and CONT.

INPUT #I%
```
100 INPUT #0,"Your name ... ",N$
```

The INPUT #I% statement is similar to the INPUT statement except that it requests input from the named Channel I%, Channel 0 being your terminal. No prompting '? ' is issued, and (as with the INCH$(I%) statement) a final PRINT statement may be necessary to prevent succeeding message-strings from overlaying the current one.

See also INCH$, INPUT, INPUT LINE, BREAK and CONT, and the Special Sections on Disk File I/O and Channel #0.

INPUT LINE
```
100 INPUT LINE A$
200 INPUT LINE A$(3)
```

The INPUT LINE statement inputs an entire line, including commas, into the named String-Variable. No message-string can be displayed, and only one variable-name may be specified. INPUT LINE outputs a simple '? ' prompt, and, unlike INPUT, will accept a CR as a valid response, setting the named variable to a NUL string.

See also INPUT, INPUT #0, INCH$, BREAK and CONT.

```
INSTR(I%,J$,k$)   100 X% = INSTR(5,J$,"there")
                  100 K$ = "there": X% = INSTR(2,J$,K$)
```

INSTR searches for string K$ **IN STR**ing J$, commencing with the character at position I%. If found it returns an integer-value identifying the **absolute** position (not relative to I%) at which K$ commences in J$, otherwise a value of 0 is returned.

```
INT(I)          100 J = INT(I)
                100 J = INT(123.456)
```

In each of the examples, J takes on a value equivalent to the integer which is not more positive than the argument in parens. Sometimes referred to as the "floor" of the argument. Some examples follow :

```
   INT(5)  =  5    INT(.5)  =  0    INT(5.6)  =  5   INT(0)  = 0
   INT(-5) = -5    INT(-.5) = -1    INT(-5.6) = -6   INT(PI) = 3
```

```
KILL            100  KILL "CHESS"
```

This statement deletes the file named "CHESS.DAT" from the disk. The default extension is .DAT, and the default drive the Working-Drive. No "ARE YOU SURE" prompt is issued, the delete action being immediately carried out, so you are advised to exercise extreme care in its use.

```
LEFT$(I$,J%)    100 H$ = LEFT$(I$,5)
                100 I$ = "Hello!": PRINT LEFT$(I$,3)
```

The LEFT$ function returns a string equivalent to the J% leftmost characters of the named string I$. Example 2, for instance, displays 'Hel', the 3 leftmost characters of "Hello!". J% **must** lie in the range 1 - 32767 in order to avoid an error-message.

See also MID$ and RIGHT$

```
LEN(I$)   100 I$ = "Hello there": PRINT LEN(I$)
```

The LEN function returns an Integer count of the number of characters in the named string. Thus the example displays 11, the total length of I$.

```
LET       10 LET X = 1.5
          10 X = 1.5
          10 X% = X% + 1
          10 X$ = "RBASIC"
          10 X% = Y
          10 X = Y% + .5
```

The LET statement assigns a value to a variable, whether Integer, FP or String. Generally, the word 'LET' is omitted and the statement used in its implied form. Thus, examples 1 and 2 are equivalent. Example 3 increments the value of X% by 1; example 5 assigns to X% the integer value of Y, while example 6 assigns to the FP variable X the value of the integer Y% incremented by 0.5.

LINE See INPUT LINE

LIST
```
LIST
LIST 50
LIST -50                 (undocumented feature of XBASIC)
LIST 50-                                                  **
LIST 50-100
LIST 50,100-150,200                                       **
```

The LIST command displays a listing of the named Line-Numbers. In the above examples, the first LISTs the entire program, while the second LISTs only Line 50. Example 3 LISTs all lines from the beginning of the program up to and including Line 50, and example 4 the opposite function, namely, all lines from Line 50 to the **end** of the program are LISTed. Example 5 restricts the LISTing to Lines 50 to 100, and the final example produces a compound LISTing composed of Line 50, Lines 100 to 150, and Line 200.

Unlike XBASIC, LIST may also be used to display a LISTing of a compiled (or .BAC) file, **for informative purposes only.** *

LOAD LOAD "CHESS"

The LOAD command will load the named text-type file from disk into RBASIC. The File-Name should be in quotes, and defaults to the Work-Drive and to a .BAS extension. Text files, produced by TSC's EDITOR or other word-processors, may also be LOADed in this way.

LOG(I) L = LOG(2.5)

This function calculates the 'natural' logarithm of the argument enclosed in parens to the base 'e'. To translate to logarithms in other bases the following formula should be used, where 'b' is the desired new base :

Log**b**(I) = LOG(I)/LOG(b)

Thus to find the common (base 10) logarithm of the number 123, the following formula is used :

LOG**10**(123) = log(123)/log(10)

```
LSET      100  OPEN "MYFILE" AS 1
          200  FIELD #1, 8 AS I$, 10 AS J$
          210  GET #1: PRINT I$,J$
          220  LSET I$ = "Hello": RSET J$ = "there"
          230  PUT #1, RECORD 1: CLOSE 1
```

It was mentioned in the definition of 'FIELD' that the contents of the I/O Buffer may not be changed by using LET (or implied LET). Specified strings in the Buffer may, however, be changed with the LSET or RSET statements. In the example above, at Line 210, Record #1 loads into the I/O Buffer and its contents are displayed. Line 220 then modifies I$ by changing it to "Hello" (**left**-justified by LSET and padded on the right with 3 SPACEs), and also modifies J$ by changing it to "there" (**right**-justified by RSET and padded on the left with 5 SPACEs). Line 230 then stores this new data in the Disk-File MYFILE.DAT before closing it off.

LSET may also be used with regular (non-FIELDed) strings in a program. So

```
100  I$ = "               "          (15 SPACEs)
300  J$ = "Hello there": LSET I$ = J$: PRINT I$
490  Cursor re-positioned to start of previous message
500  LSET I$ = "I'm OK!": PRINT I$
```

defines I$ as being 15 characters in length in Line 100. Line 300 displays J$, left-justified in I$ and padded with 4 SPACEs to the right. Now Line 500 overwrites the previous message with the shorter "I'm OK!", and, by padding with SPACEs, leaves no trace of the extra characters of the earlier message.

MID$(I$,J%) 100 I$ = "Hello there": PRINT MID$(I$,5)

If MID$ is used with only two arguments enclosed in parens it returns a string commencing at character-position J% of the named string I$. J% should lie in the range 1 - 32767. The example would return 'o there'.

MID$(I$,J%,k%) 100 I$ = "Hello there": PRINT MID$(I$,5,4)

If MID$ is used with three arguments enclosed in parens it returns a string K% characters long commencing at character-position J% in the named string I$. Thus the example returns the string 'o th'.

```
MOD          100  A% = -17: B% = 5
             150  X% = A%/B%: Y% = RESIDUE
             200  PRINT X%,Y%
```

RBASIC does not have a MOD statement per se, but does provide the residue of an Integer-Divide operation by assigning its value to the variable RESIDUE. Line 150 therefore calculates the quotient and assigns its value to X%, and also the residue (in this case -2) to Y% The residue may be obtained at any time after an Integer-Divide operation.

MON MON

The MON command exits RBASIC and transfers control to the System monitor. Memory locations may now be examined (and prudently changed, if necessary). Re-entry into RBASIC's Warm-Start location may then be made from GMXBUG by entering 'J 3', or if your monitor does not offer this facility, control should be returned to FLEX and 'JUMP 3' executed. In both cases, any RBASIC program resident in memory will be retained intact.

NAME(variable-name)

```
PRINT NAME(A)                                              **
POKE NAME(M$),65
DPOKE NAME(A%),66*256 + 67
DPOKE NAME(BC),65*256
DIM A%(5): POKE NAME(A%(0)),66
DIM A%(4,4): DPOKE NAME(A%(0,0)),66*256 + 67
```

This function returns the address (in decimal) where the variable-name is stored. In the case of subscripted variables, it is necessary either for the desired variable to be DIMensioned, or for the program to have been RUN past the appropriate DIM statement, before calling NAME.

Example 1 merely returns the address where the variable-name 'A' is stored. The remaining examples demonstrate how to use NAME as an Editor to change a variable-name globally.

Example 2 changes the name M$ to A$ (where 65 is the decimal ASCII-value of 'A'), 3 changes A% to BC%, and 4 changes BC to A. Note the use of the 256-multiplier to position the first (or only) letter of the name in the most-significant character-position in these two examples. When changing a single-character name to another single-character, this will not be necessary, though POKE should then be used instead of DPOKE.

Example 5 shows how to change the subscripted variable A% to B%, where the RBASIC program in which it resides has not yet been RUN. If it has actually been RUN, the DIM instruction should be omitted. Similarly, example 6 shows how to change a doubly-dimensioned variable's name. Although the dimension '0' is used in examples 5 and 6 (for ease of use), it may actually be **any** dimension within the range of the DIM statement.

Be **very careful** not to change a variable's name to one that's already in use in your program!

See PTR

NEW

```
NEW
```

The NEW command deletes the entire RBASIC program from memory, leaving RBASIC ready to accept a fresh program.

NEXT

```
200  NEXT I%
```

The 'NEXT' statement defines the boundary of the loop enclosed between it and the corresponding 'FOR' statement, that is the 'FOR' defining an index-variable of the same name as that following the word 'NEXT'. 'NEXT' increments (or decrements) its index-variable by the amount specified (or implied) in its matching STEP-size. Control then passes back to the 'FOR' statement for checking whether the defined limit has been exceeded.

See also FOR

NOT

```
PRINT NOT 5
100  IF NOT(Q$ = "Y" OR Q$ = "y") GOTO 300
```

The NOT logic operator performs a bit-wise complement operation on the Integer value of numbers, or on the truth-values of conditional clauses. Thus example 1 produces a result of -6, as the complement of 5 (0101) is 1010, which needs 6 (0110) added to bring the result to 0000, with a carry. The 'NOT' of an Integer will always be equal to '(the negative of the number) - 1'.

In example 2, were it not for 'NOT' the program would branch to Line 300 if Q$ were equal to "Y" or "y", but 'NOT' complements the complete truth-value so that it will now branch if Q$ is anything **but** "Y" or "y".

ON

```
  50  ON I% GOSUB 100,200,300, ...
  50  ON I% GOTO 100,200,300, ...
  50  ON ERROR GOTO 1000
1100  ON ERROR GOTO 0
1100  ON ERROR GOTO
```

Each of the first three lines above demonstrates a different use of the 'ON' statement. In Example 1, Line 50 branches to an appropriate subroutine, depending on the value of I%. Upon encountering a 'RETURN', control is passed to the line following Line 50, therefore ON-GOSUB should be the last statement on the line.

Example 2 is similar, except that control is passed **directly** to the named line, depending on the value of I%.

In both cases, if I% has a value of 0 or exceeds the number of named lines to which to branch, an error results.

Example 3 traps errors which would otherwise abort the program and display an error-message before returning to the 'RBASIC' prompt. You may require the program to process certain errors, and continue execution, rather than grinding to a halt. In the example above, should an I/O error occur (Error-Number < 50) in any line following the 'ON ERROR' statement, control would pass to Line 1000 to carry out the instructions resident there. See RESUME for details on how to exit the error-handling routine.

Examples 4 and 5 are functionally equivalent, simply cancelling the preceding 'ON ERROR GOTO XXXX', thus allowing RBASIC to handle any future errors in the normal way, unless another 'ON ERROR GOTO XXXX' occurs later. Should either of these lines occur **within** an error-handling routine, it will cancel the effect of the routine and make RBASIC behave as though the routine had never been called.

See ERROR-TRAPPING, RESUME

OPEN

```
100  OPEN NEW "MYFILE" AS 1
100  OPEN OLD "YOURFILE" AS 3
100  OPEN "JOESFILE" AS 5
```

The OPEN statement is used to create a new file for data storage, or to access an already existing file for either a WRITE or a READ operation. See DISK-FILE INPUT/OUTPUT on pages 7 - 12 for full details.

See also CLOSE

OR

```
PRINT 3 OR 5
100  IF Q$="Y" OR R$="P" GOTO 300
```

The OR logic-operator performs a bit-wise **inclusive-OR** operation on the Integer value of numbers, or on the truth-values of conditional clauses. Thus example 1 produces a result of 7 (111) when ORing 3 (011) with 5 (101) because a 1-bit occurs in every bit-position of either 3 or 5 or both.

Example 2 causes program-flow to transfer to Line 300 if either or both of the conditional clauses 'Q$="Y"' or 'R$="P"' were true, otherwise it falls through to the next line.

PDEL

```
PDEL                                                    **
PDEL -50                                                **
PDEL 50-860                                             **
PDEL 50-                                                **
```

PDEL performs a block-delete of the specified lines. Obviously, the standard form of deletion, say '50(CR)', should be used to delete a single line, or several widely separated lines.

PDEL is a prompting-delete, and upon being invoked will prompt for a response of Y/N/CR to each line displayed. 'Y' indicates 'Yes' and thus deletes the displayed line, while 'N' (No) bypasses the currently displayed line and moves on to the next - if there is one. A response of 'CR' aborts the deleting session and returns control to RBASIC.

To prevent accidental destruction of the Variable-Stack or other registers, **the final line of the program cannot be deleted with PDEL.** The standard method **must** be used to delete this line.

Example 1 begins prompting at the lowest program line-number, and proceeds through to the last line but one. Example 2 commences with the lowest line-number and proceeds through to Line 50, if it is not the final line. Example 3 prompts for deletion of the block commencing with Line 50 and continuing through to Line 860, while the final example commences at Line 50 and proceeds through to the end of the program **EXCEPT FOR THE FINAL LINE.**

If more than one deletion-block is specified (as with LIST, where this is permissible), only the first such block will be acted on. Thus in the line :

```
    PDEL 10-200,500-700
```

PDEL will act only on the block 10 - 200, and then return control to RBASIC.

PEEK(I)

```
100 X% = PEEK(HEX("E010"))
100 A = HEX("E010"): X% = PEEK(A)
```

This function is the opposite of POKE, and returns the Integer value of the byte at address (I). The maximum value of address I is 65535 ($FFFF), and the maximum value returned will be 255 ($FF).

See also POKE, DPEEK and DPOKE

PI

```
PRINT PI
```

This returns the FP value of PI to 17 digits. If the DIGITS statement has previously set DIGITS 5,3 the example displays 3.142, although PI is stored internally as 3.1415926535897933.

POKE

```
100 POKE HEX("E010"),29
100 A = HEX("E010"): POKE A,X%
```

The POKE statement performs the opposite function to PEEK, storing asingle byte at the decimal address named. Maximum value of the address is 65535, or $FFFF, and the maximum value POKEd to that address should be 255, or $FF.

See also PEEK, DPEEK and DPOKE

```
POS(I%)   100 PRINT POS(0)
```

The POS function returns the current column-position of the named Channel, commencing at 0 for the leftmost margin. The example displays the current column-position of your screen-cursor.

```
PRINT     100 PRINT
          110 PRINT A;B
          120 PRINT A,B
          130 PRINT "Hello!"
          140 PRINT "Area =";A * B
```

The PRINT statement displays information on your terminal. Example 1 above outputs a CR/LF, as no data follows the PRINT statement. Example 2 displays the current value of variable A, followed immediately by the current value of variable B, while example 3 displays the same variables, but with 'B' TABbed over to the next internal TAB of 16 columns. The maximum length of the line is determined by the FLEX variable 'WIDTH', where a width of '0' signifies indefinite length.

Example 4 displays the message "Hello!", and the final example the message 'Area =' followed immediately by the value of the product (A * B).

See also '?' and the Special Section on Disk I/O and Channel #0

```
PRINT USING  100 PRINT USING '###.##',123.456
             100 PRINT USING '\  \\  \',"Hello","everyone"
             100 A$ = "##.# is the square root of ###":
                 PRINT USING A$,SQR(I),I
```

PRINT USING is a much more powerful form of the PRINT statement, permitting the format of the printed output to be controlled by means of certain special characters, such as the '#' and the '\' in the examples above. The formatting-string may be enclosed between single quotes as in Examples 1 and 2, or be pre-defined as in Example 3.

The separators between the items in the list to be printed are normally ignored by PRINT USING (whether a ';' or a ','), unless one of them occurs at the very end of the **format-string**, when it will be interpreted as a positioning instruction for the next PRINT statement to occur. For this reason, the formatting-field **must** end with a format-character, and not with text. Thus the following is invalid:

```
100 PRINT USING '###.## cents',123.456
```

and should be expressed as:

```
     100 PRINT USING '###.##',123.456;: PRINT "cents"
```

where the ';' ensures that 'cents' is displayed immediately after '123.46'

The special formatting-characters are :

POUND SIGN is used to format a numeric field. Thus the command-line

```
     PRINT USING '###.##',123.456
```

prints the result 123.46 (rounded to 2 decimal places). If the number to be formatted is too large for the formatting-field, it will be preceded by a '%' sign and printed as though there were no format-field defined. If the fractional part of the number is too small, it will be padded to the right with '0's, and if it is too large it will be rounded. If, as a result of being rounded, it becomes too large for the format-field, it will again be preceded by a '%' sign and printed **in its rounded form.** Thus PRINT USING '##.#',99.95 is printed as %100.0.

The format-field will be terminated if any character other than plain-text, a comma, period or up-arrow occurs in the field.

DOLAR SIGN is used to format money fields, preceding the cash amount with a '$' sign. The first TWO characters of the field should be dollar-signs, one of which counts as an additional character in the format field. Thus:

```
     PRINT USING '$$##.##',123.456    would print   $123.46
```

The rules governing oversized numbers are the same as for ordinary numeric fields. Except that negative amounts (unless they have a trailing minus-sign) cannot be used with the leading dollar-sign.

TRAILING MINUS is used to represent negative numbers when a format-field contains a leading dollar-sign or an asterisk-fill. The example:

```
    PRINT USING '$$##.##-',-12.345            displays  $12.35-
```

ASTERISK is used to fill with asterisks the leading blanks of a numeric or money field to prevent the amount specified from being easily altered. Just as with the dollar-sign, **two** asterisks are necessary to specify this format, one of which is counted as part of the field. At least one asterisk must be printed out, and if the numeric or money field is too large to allow this, the amount will be displayed unaltered and preceded by a '%' sign. Thus:

```
     PRINT USING '**##.##', 12.34           displays as   **12.34
```

In the case of money fields, a single dollar sign (not counted as a character) should precede the asterisks in the format-field. Thus:

```
     PRINT USING '$**##.##', 1.23        displays   $***1.23
```

COMMA if inserted in a numeric field to the left of the decimal-point causes commas to be displayed every 3 places. **Embedded commas count as part of the numeric field.** If a comma occurs before the numeric field or after the decimal point it is considered to be a literal, and will be printed as such. Again, if the numeric field overflows the format-field a '%' sign will precede the number, which will be printed as far as RBASIC is able to fill the format-field.

```
  PRINT USING '#,,,#.##', 1234.567           displays as   1,234.57
  PRINT USING '#,###.##', 1234.567      also displays as   1,234.57
  PRINT USING '###,###,###.##', 1.23456E6   displays   1,234,560.00
```

EXCLAMATION MARK is used to format a single-character field, thus :

```
     PRINT USING '! ! !', "University of","British","Columbia"
```

displays U B C, the first character of each of the string-fields.

BACK-SLASH is used to format a string-field of 2 or more characters, the size of the field being determined by the number of characters between a pair of back-slashes, which also contribute to the field-size. Normally you would fill the space between the slashes with numbers, to make the size of the field readily discernible, as shown below:

```
     PRINT USING '\12\\123\', "Hello", "everyone"
```

displays as 'Hellevery', being 4 characters from the first string-field and 5 from the second.

UP ARROW is used to signify Scientific Format **for numeric fields only**. Exactly 4 up-arrows must trail the numeric format-field to represent the 'E+XX' structure of Scientific Notation. Thus, assuming that the trigonometric functions have been set to DEGREES, the following

```
     PRINT USING '#.####^^^^', SIN(30)       displays   5.0000E-01
```

PTR(variable-name)   PRINT PTR(I%)

This function returns the address (in decimal) where the value of the named variable is stored. In the case of an Integer or Floating-Point variable, the address is the actual address where the variable's current value is stored.

If the variable is a String-Variable, the value returned is the address of a String-descriptor, which consists of 4 bytes, the first 2 bytes of which are the actual address of the string, while the next 2 bytes indicate the length of the string. The string itself may contain any character up to the full 8-bit code, and is not restricted to the 7-bit ASCII code.

See NAME

PUT

```
100  OPEN "MYFILE" AS 1
110  GET #1, RECORD 3
120  PUT #1              or   120 PUT #1, RECORD 3
```

The PUT statement is used in Record I/O to store a specified Record back to the Disk-File on the named Channel. Because the Record-Number is automatically incremented each time it is accessed, Line 120 actually stores the contents of the I/O Buffer into Record 4. If you desire to save back to the **same** Record-Number, this should be specified - as in the alternative Line 120.

PUT is also used to extend the number of Records in a File, by PUTting to a Record-Number higher than the current maximum in the File.

See also FIELD, GET, RECORD, and the Special Section on Disk I/O

READ

```
10 READ A%,B,C$
```

The READ statement reads the DATA item currently pointed to by the DATA-pointer, and assigns this item to the named variable. The example above READs item 1 and assigns its value to A%, then READs item 2 and assigns it to B and finally READs item 3, assigning it to C$. If the data-type read does not match with the variable-type (for example, if the program attempts to assign the string "Hello" to the variable A%), or if there are insufficient items to satisfy the READ statement, an appropriate error-message will be generated.

See also DATA, RESTORE

REM

```
100 REM
100 REM CHECK VALID 'PAWN' MOVE
```

The REM statement inserts comments throughout a program. It is good practice to indent REMs by only one SPACE, and the rest of the program by at least two. Normally REMs are used at the start of a program to identify the program itself, and throughout the program to indicate the function of the block of lines immediately following. RBASIC ignores everything which follows the REM statement, therefore it should be the only, or last, statement in a program-line.

RENAME   10 RENAME "CHESS","CHESS1"

This statement RENAMEs a disk file, the example causing the file "CHESS.DAT" to be RENAMEd to "CHESS1.DAT". File names default to the Work-Drive and to an extension of .DAT.

RENUM

```
+RENUM
+RENUM 100
+RENUM 200,5
```

The RENUM command is entered in response to the 'RBASIC' prompt, and must be preceded by a '+' to indicate to RBASIC that a Utility command is being called. Example 1 renumbers the Lines in an RBASIC program in a default mode, i.e. with the first Line-Number being 10 and an interval of 10 between successive lines. Example 2 begins renumbering the lines with 100 and a default spacing of 10 between lines, while Example 3 commences numbering with 200, with an interval of 5 between lines.

Unlike XBASIC, RBASIC correctly handles the renumbering of ERL statements of the form "IF ERL = 123 ... ". **

Also, unlike XBASIC, RBASIC will, during the renumbering process, identify all references to non-existent line-numbers by tabulating the new line-number and all such references existing on that line. **

RESIDUE   see MOD **

RESTORE

```
10  RESTORE
10  RESTORE 100
```

The RESTORE statement restores the DATA-pointer either to the first occurrence of a DATA item in the program (as in example 1) or to a specific DATA line-number (as in example 2). RESTORE thus behaves as a controllable RESET for the DATA-pointer.

See also DATA, READ

RESUME

```
100  IF ERR = 34 THEN RESUME
100  IF ERL = 200 THEN RESUME 300
```

The RESUME statement pairs with 'ON ERROR GOTO' in much the same way that RETURN pairs with GOSUB. The first example causes program-execution to return to the **start** of the line in which error-number 34 occurred, while the second, if an error occurred in Line 200, returns control to Line 300. You should exercise extreme care under these conditions to ensure that control is not returned to a different level of subroutine-nesting, otherwise the Stack-Pointer will be out of step and results will be unpredictable!

RESUME should always form part of error-handling routines, and is the only valid form of return. **DO NOT USE 'RETURN'.**

See also 'ON ERROR GOTO'

RETURN
```
100 RETURN
100 IF I = 5 THEN RETURN
100 IF I = 3 THEN RETURN ELSE PRINT"OK": rest of program ..
```

This statement exits from a subroutine (called by GOSUB or ON-GOSUB) and returns program-control to the statement (or Line, in the case of ON-GOSUB) immediately following the GOSUB statement. Apart from an IF-THEN-ELSE type of statement (see example 3 above) RETURN should be the last, or only, statement on a line.

See also GOSUB, ON-GOSUB

RIGHT$(I$,J%) `100  H$ = RIGHT$("Hello",3)`

This statement forms a character-string equivalent to the rightmost J% characters of I$. Thus, in the example, H$ is equal to "llo", the rightmost 3 characters of "Hello". J% is restricted to the range 1 - 32767. If J% is equal to, or greater than, the length of I$, the whole of I$ will be returned.

RND(I)
```
100  J = RND(0)
100  J = RND(3)
100  J = RND(-3)
```

RND returns a random number in the range 0 up to, but not including, 1. Example 1, where the argument in parens is '0' returns a new random number each time it is called, and is the normal way in which RND is used.

Example 2 (positive argument) freezes the random-number generator and returns the last random-number generated.

Example 3 (negative argument) 'seeds' the number-generator with a specific number, to begin a new series of random numbers. Each time the same negative-number is used as a seed, however, the number-generator will be returned to the same initial number, and the same series restarted.

To generate fractional numbers lying between known limits, the following formula should be used:

```
Random FP = (U - L) * RND(0) + L
```

where 'U' is the upper-limit (which will itself never be generated) and 'L' the lower-limit.

The following formula generates **integers** between specified limits, where 'U' and 'L' have the same definitions as above, the additional '1' guaranteeing that the upper-limit **will** be included in the range of numbers generated :

```
     Random Integer = (U - L + 1) * RND(0) + L
```

RSET   See LSET for explanation and example.

Whereas LSET **left**-justifies string I$ into string J$, RSET does the exact opposite by **right**-justifying I$ into J$ and padding on the **left** with the necessary amount of SPACEs.

RUN

```
RUN
RUN "CHESS"
```

The RUN command in Example 1 causes the program currently in memory to begin execution, after it initialises all variables to zero, and the DATA pointer to the first-occurring DATA statement in the program.

The second example first LOADs the program 'CHESS.BAC' from disk, and begins immediate execution. The only other way to LOAD a compiled (.BAC) program from disk is to enter RBASIC CHESS (or other program-name) in response to the FLEX prompt '+++', which LOADs RBASIC first, followed by the program itself, and finally an automatic RUN command.

SAVE

```
SAVE "CHESS"
```

The SAVE command saves to disk the .BAS file currently in memory, **deleting and replacing any file of the same name presently on disk.** The extension defaults to .BAS and to the Work-Drive. Files SAVEd in this way may later be processed with any FLEX Editor or other Word-Processor. The File-Name should be in quotes.

SCALE

```
SCALE
SCALE 2
```

The SCALE command sets the number of digits to be preserved to the right of the decimal point, to a maximum of 6, with a Scale-Factor of 0 turning off the SCALE feature. The SCALE factor should be set before a program is LOADed or typed in, as it is not possible to change it afterwards.

When a program is COMPILEd to disk the Scale-Factor in use at the time is also saved with it, and is automatically re-instated when the program is later reloaded into memory for execution.

Example 1 above displays the current Scale-Factor, while Example 2 sets the Scale-Factor to 2. This tells RBASIC to SCALE all Floating-Point numbers by a factor of 10^2 **and then round to the nearest whole**

number of course, the higher the Scale-Factor, the smaller the actual dynamic-range of the numbers becomes. To demonstrate one of its uses, consider the following :

```
   10  FOR I% = 1 TO 10000: J = 0.01 + J: NEXT I%
   20  PRINT 100 - J
```

When this short program is RUN, a result of 1.77813319624E-12 is displayed. This occurs because '0.01' cannot be converted to an exact binary value internally, and so, after 10,000 addition-operations the cumulative error amounts to a small but possibly significant amount. Now, if NEW is typed, followed by SCALE 2, and the short program re-typed and RUN again, the result is correctly displayed as '0'. The testprogram requires a Scale-Factor of 2 because the amount by which 'J' is incremented each time is equal to 0.01, which is now SCALEd to 1. As '1' **can** be EXACTLY represented in Floating-Point notation there is no round-off error to accumulate, and so we end up with a precise result.

SGN(I)
```
100  PRINT SGN(-123.456)
```

The SGN function returns a number indicating the 'sign' of the named variable. A '1' is returned if the number is positive, '0' if it is zero, and '-1' if it is negative. Thus the example displays '-1'.

SIN(I)
```
10  PRINT SIN(1.2)
10  PRINT SIN(I - PI / 6)
```

The function SIN calculates the trigonometric SINE of the argument enclosed in parens, which is expressed in radians or degrees, depending on the setting of the DEGREES/RADIANS switch.

See DEG

SPC(I%)
```
100  PRINT SPC(30)
100  PRINT #0, SPC(30)
```

The SPC function prints the specified number of SPACEs at the current cursor-position, or at the print-head position if directed to a Printer. It may only be used in PRINT statements, and its argument may not exceed 255.

SQR(I)
```
10  PRINT SQR(16)
10  PRINT SQR(I - 5)
```

This function returns the square-root of the number enclosed in parens. A negative argument produces an error-message.

STOP
```
100  STOP
```

The STOP statement is usually inserted in a program for debugging purposes, as it causes program execution to be suspended, with a message displayed informing you at which line the STOP was encountered. Thus:

```
    STOP at Line 50
```

At this point the values of certain variables may be examined or changed, an entry of CONT causing the program to resume **at the line following that at which it STOPped.** For this reason STOP is usually put at the end of a line, or more commonly inserted in the program on its own temporary line, as shown above. If the program itself is changed in any way (by adding or deleting a line, perhaps), it will **not** resume execution with CONT.

STR$(I)

```
10  I$ = STR$(I)
10  I = 123.4: I$ = STR$(I)
```

STR$ is the opposite of VAL, converting a numerical expression 'I' to a string of ASCII characters corresponding to each character of I. In example 2, I$ is equal to "123.4". The conversion includes '-' or implied '+', and also honours the DIGITS statement.

SWAP

```
100  IF A% > B% THEN SWAP A%,B%
100  SWAP A$(I%),A$(I% + 2)
```

The SWAP statement exchanges the values of the two specified variables, mainly in data-sorting applications. The variables must be of the same type, and be memory-resident - that is, **they cannot be part of a virtual array on disk.** String-sorting is especially fast as only the pointers to the strings are exchanged, not the strings themselves.

TAB(I%)

```
100  PRINT TAB(30); "Hello!"
100  PRINT #0, TAB(30); "Hello!"
```

The TAB function may be used only as part of a PRINT statement, and in the examples it moves the cursor (or print-head in the case of a Printer) to column 30. No action is taken if the cursor is already at, or past, this position. I% is, of course, restricted to a maximum value of 255.

TAN(I)

```
10  PRINT TAN(1.5)
10  PRINT TAN(I + .3)
```

The function TAN calculates the trigonometric TANGENT of the argument enclosed in parens and expressed in degrees or radians, depending on the setting of the DEGREES/RADIANS switch.

See DEG

THEN See IF.

```
TRACE      TRACE 1                                                    **

           100 TRACE 1         (or TRACE 255)                         **
           200 TRACE 0                                                **
           300 TRACE 50                                               **

           100 IF X > 7 THEN TRACE 1                                  **
```

The TRACE statement turns the TRACE feature of RBASIC on or off. 'TRACE 1' (or 'TRACE 255') causes RBASIC to display in square-brackets the Line-Number of each line executed during a program RUN. This is a very useful feature for debugging a program, especially as multiple TRACEs are displayed per screen-line. 'TRACE 1' is normally cancelled by 'TRACE 0', though NEW has the same effect (besides cancelling the program currently in memory). The TRACE statement replaces the 'TRON' and 'TROFF' commands of XBASIC.

TRACE may also be used to trace a specified number of lines, with a minimum of 2 and a maximum of 254. Thus the statement in Line 300 above TRACEs exactly 50 lines, then automatically turns off the TRACE.

TRACE runs in Single-Step mode, halting after the display of each Line-Number. Hitting any key (except RETURN) advances program execution by one line, so that program flow may be easily TRACEd under complete control of the User. Hitting RETURN cancels TRACE-mode, and allows the program to resume normal execution.

USR(I)

USR calls a machine-language program, usually to pass a 16-bit parameter to such a program, and/or to retrieve a 16-bit value returned by this program. The data to be passed (or retrieved) is located at address MEMEND-4, which we shall call USRDAT, and the address of the machine-language program to be called is located at address MEMEND-2, which we shall call USRVEC.

It is only necessary to DPOKE the address of the desired M/L program into USRVEC, and when USR is called RBASIC will evaluate into 16-bits the argument to be passed, storing it in USRDAT. Similarly, if a result of some operation is to be returned from the M/L program it will normally be found stored at USRDAT awaiting retrieval. A prime requirement is that the program being called should preserve the value of RBASIC's Stack-Pointers, and further, it should not use up more than 256 bytes of Stack-space for its own use.

With all this in mind, this is how an argument of '8' would be passed to a Machine-Language program at $C100, which would, let's assume, compute the square of the argument passed to it :

```
   100  UV = DPEEK(HEX("CC2B")) - 2: REM Address of USR-VECTOR
   110  DPOKE UV, HEX("C100")
   120  X = 5 * USR(8)
```

and let's assume the M/L program looks something like this:

```
  MEMEND EQU $CC2B Memory end
   ORG $C100
  SQUARE FDB 0  Sqare of passed parameter

   PSHS U       Save U
   LDU  MEMEND  Get Memory end address
   LEAU -4,U    Point to USRDAT
   LDD  0,U     Get argument (in this case 8, as in Line 120 above)
   ***          Code to calculate Square of argument here
   LDD  SQUARE  Get value to be returned
   STD  0,U     and store in USRDAT
   PULS U,PC
```

Line 120 of the RBASIC program not only passes the value '8' to the machine-language program for processing, but assigns to the variable X the value 320, ie, 5 * 8^2.

Multiple USR routines.

If certain routines to be called do not require a parameter to be passed, then the parameter-passing function of USR can be used to specify to a machine-language routine which of several alternative functions to perform.

Another method, if parameter-passing is required, is to assign names to the addresses of the various machine-language routines, which are then DPOKEd into USRVEC in order to be called. Thus :

```
    100 UV = DPEEK(HEX("CC2B")) - 2: REM (address of USR-VECTOR)
    110 I = HEX("A000"): J = HEX("B000")
      . . . . . . . .
    500  DPOKE UV, I:  A = USR(8)
    510  DPOKE UV, J:  B = 3 * USR(17)
```

Line 500 then DPOKEs address 'A000' into USRVEC, passes parameter '8' to this routine for processing, and finally assigns to variable A the value returned in USRDAT. Similarly, Line 510 DPOKEs address 'B000' into USRVEC, passes the parameter '17' to the routine for processing, and finally assigns to variable B a value equivalent to 3 times the parameter retrieved from USRDAT. Of course, if the called routine were simply required to perform some task without the need of a parameter, then a statement using a dummy variable such as 'A = USR(0)' would initiate this task.

VAL(I$) `10  I = VAL(I$)`

VAL is the exact opposite of STR$, converting a numeric string (composed of numbers, '+' or '-', or decimal-point) to its numerical VALue. If the first character of the string is anything other than a valid numeric character, a value of 0 will be returned, thus VAL("Hello") = 0. Conversion of the string will cease at any point where a non-numeric character is encountered, thus VAL("12H45") = 12, conversion ceasing at the 'H'.

XOR

```
PRINT 7 XOR 5                                           **
80  IF Q$ = "Y" XOR R$ = "Y" GOTO 100                   **
```

The XOR Logic-Operator performs an Exclusive-OR operation on the expressions linked by it. In Example 1, a bit-wise operation is carried out on 7 (111) and 5 (101), resulting in a '1' bit if **either** of the corresponding bits (but not both) is a '1'. The result in this case is 2 (010), as it is only in the centre-bit position that one term has a 1-bit while the other has a 0-bit. In Example 2, the truth-values of (Q$="Y") and (R$="Y") are XORed, so that only if one condition or the other is TRUE will control be transferred to Line 100. Otherwise, if both conditions are FALSE or both are TRUE, the program falls through to the following line.

When used with truth-values, XOR should be restricted to coupling **two** conditional clauses only. Because XOR is also a “toggling" function, its use in a line such as the following can possibly lead to undesired results:

```
10 IF Q$ = "Y" XOR R$ = "Y" XOR S$ = "Y" GOTO 100
```

as XOR evaluates as TRUE if an **odd** number of the conditional clauses is true, and as FALSE if an **even** number is true. Remember that '0' is an even number!!

RBASIC LINE-EDITOR

INTRODUCTION

The built-in Line-Editor is one of the more powerful features of RBASIC and puts a whole new range of program-development tools in your hands. In this discussion, the various editing features are referred to by name, rather than by their corresponding Control-Codes, as these are customisable and will vary from system to system.

As with RBASIC, the Editor may be used either in **IMMEDIATE** mode or in **PROGRAM** mode. In either mode you may cursor right or left, though movement is restricted to the length of the line being edited.

IMMEDIATE MODE

In the simplest case, let's assume that you have attempted to LIST 100-500 to home in on a section of interest, but were unable to halt the LISTing in time (using the ESC key). In this event, hitting the LINE-FEED key **feeds the line** back to RBASIC for a repeat execution. On the other hand, the RECALL-LAST-LINE key (hereafter abbreviated to RECALL) clears the screen and recalls the LIST instruction, either for immediately feeding back to RBASIC or for possible editing before so doing. You should standardise this key on ^R, if possible, to conform both to the TSC EDITOR's REPEAT function and to Micronics Research's adaptation of FLEX. A RECALLed line is always in OVERLAY mode (hence the absence of an OVERLAY key).

More usually, however, the situation will arise where you have typed in a line as part of a program, only to have it rejected by RBASIC for, let's say, a 'MISSING QUOTE IN STRING' or some other syntactical error. Here again you would use the RECALL key to recall the line for editing by means of INSERT, DELETE or a simple overlay. It should be pointed out here that in all cases where a line is recalled for EDITing (whether in IMMEDIATE or PROGRAM mode) a CR will be placed at the end of the recalled line **only in the INPUT-Buffer**, otherwise a RECALL would immediately re-feed the line back to RBASIC. If you elect to extend the line by positioning the cursor at the end of the displayed line, over this internal CR, and then commence typing, the CR is automatically moved out ahead.

Alternatively, if you extend the line by using the INSERT key (even at the very end of the line), the internal CR is similarly moved out to the right as extension occurs. In both cases, feed the line back to RBASIC by hitting the LINE-FEED key, which henceforth is interpreted as 'FEED LINE to RBASIC'. Similarly, if you choose to cursor to some intermediate point in the line to INSERT or DELETE characters, the line-length is adjusted automatically (still retaining the internal CR). Consequently, it is not necessary to move back out to the end of the line, a simple LINE-FEED causing RBASIC to read the Input-Buffer as though you have just typed in the entire line, terminating with its internal CR entry.

About the only time you need to use CR for returning an edited line to RBASIC is when a line needs to be truncated. In this case, set the cursor to the cut-point (normally over a colon) and enter CR to terminate the line at this point.

PROGRAM MODE

This mode is invoked when you wish to EDIT an already existing program-line. In this case, an entry of EDIT 50 clears the screen and brings up Line 50 for editing, with the cursor sitting at the start of the line. Again keep in mind that as far as RBASIC is concerned, the displayed line **has just been typed in by you.** Therefore, if you edit the line in any way (remember the built-in CR at the end of the line), you should feed the line back to RBASIC with the LINE-FEED key. Be aware, however, that if this EDITed line is rejected by RBASIC for an error of some kind, the original Line 50 is retained **unchanged** in the program. Therefore, if the error needs to be corrected, you should RECALL the line for further editing with ^R. A second entry of 'EDIT 50' would only recall the original program-line 50 for EDITing, and your earlier set of changes would have to be carried out again.

As we did with the main section on RBASIC, descriptions of the various key functions are arranged alphabetically.

CANCEL Normally ^X

Use this key at any time during an EDIT session to abort EDIT mode and return to RBASIC. A '^X' will be displayed on the screen, and control returned to RBASIC as though EDIT had never been invoked, leaving the source-program unchanged.

DELETE

Deletes the character immediately beneath the cursor, except for the final CR at the end of the line being edited.

EDIT

This is the only EDITing operation which does not reside in a single-key, but must be typed in in response to the RBASIC prompt. It must always be followed by the program line-number to be EDITed, thus 'EDIT 100'. A final entry of CR clears the screen and recalls Line 100 (if such a line exists) for EDITing by you.

You should not attempt to EDIT a compiled (.BAC) program-line, as the results will be entirely unpredictable, possibly disastrous. You should work only on source (.BAS) files!!

EXPRESS LEFT/RIGHT Normally the HOME-UP key.

This key alternately switches the cursor between the left-hand and the extreme right-hand of the line being EDITed. The first move will always be to the left (or HOME position), and, provided no other operation intervenes, a second EXPRESS sends the cursor to the extreme right, a third EXPRESS back left again, and so on.

INSERT

This key allows you to insert a character to the immediate left of the current cursor position. The line automatically expands to the right, with a SPACE inserted in the opened-up position and the cursor located at this point ready for typing in a new character. If, for example, a five-letter word is to be INSERTed you will find easier to hit INSERT five times, and then begin entry of the desired word, rather than INSERT, char, INSERT, char ... etc.

LINE FEED

Standardised on the normal keyboard. This key should now be read as 'FEED LINE to RBASIC'. It feeds a newly-edited line back to RBASIC **irrespective of the cursor-position in the line.** If not used in EDIT mode, it feeds to RBASIC the last line typed by you at the keyboard. It may thus be used for repeated execution of a given command line.

MELD

The MELD key, when a line is displayed for EDITing, appends the next line in the program to the end of the displayed line. A colon (:) is automatically inserted as a separator. If the line being appended is indented in its original form by one or more SPACEs, it is offset from the colon by a single SPACE only. If the original has no SPACEs between its Line-Number and the Line's data, no SPACE is inserted between the colon and the ensuing data either. The line which is appended will also remain unchanged in the source program in its original location, for later deletion in the normal way. You may hit MELD more than once in succession, causing successive program-lines to be appended each time it is operated.

OVERLAY

A line recalled for EDITing, whether in IMMEDIATE-MODE or PROGRAM-MODE, is **always** in OVERLAY mode. Characters typed at the keyboard are displayed and recorded internally at the current cursor-position.

RECALL Normally ^R

This key RECALLs for execution or EDITing the last-entered line typed in by you. Remember that a line being EDITed is regarded by RBASIC as having been manually entered by you!!

SPLIT

This key SPLITs a line at the current cursor-position (normally over a colon). The portion of the line to the left of the cursor disappears from the display and is fed back to RBASIC as a newly-entered line. The colon normally under the cursor also disappears, leaving the cursor sitting at the left of the remainder of the line, ready for entry of a new line-number. If you do not wish to retain the line-portion still on the screen, you should abort EDIT by hitting the CANCEL (^X) key.

NOTES:

In this discussion, all references to EDITing a line apply equally to the Line-Number itself, which may also be EDITed, perhaps to move a line from one location to another. Of course, if you change the Line-Number itself, it will become a **copy** of the original line (as though you had just typed it in), and will, when fed back to RBASIC, locate itself in its new position. The original line remains **unedited** in its former position in the program source-code.

See also the NAME function for a description of how to use it as a **Global Variable Name Changer!**

Throughout this entire Manual, all references to TSC imply Technical Systems Consultants, Inc., and all references to XBASIC imply TSC's Extended BASIC for the 6809.

Adapting to Your System

There are a few key addresses in RBASIC which are of interest to you. These are:

COLD $0000.
This is the COLD-START address used by RBASIC when it is first loaded into memory. If you jump to this address after leaving RBASIC, everything becomes re-initialised, and any program in memory will become unavailable.

WARM $0003.
This is RBASIC's WARM-START entry-point. Jumping to this address after leaving via MON or DOS preserves the program currently in memory, and should therefore be the normal method of re-entry.

MEMEND $CC2B/C.
These two bytes are updated by FLEX to reflect the current End-of-Memory location, and are used by RBASIC to position its Stack-Pointer. If you need room, possibly to accommodate a USR routine, then MEMEND should be set to a lower point, **before** you call RBASIC into memory. If it is done afterwards you should re-enter RBASIC via COLD to ensure that all Pointers, and so on, are correctly initialised.

MON $0006.
It may be necessary for you to reconfigure the vector-address stored here, which vectors RBASIC to the Monitor-ROM's entry-point. Currently it is set for $F814, the vector to the entry-point for both GMXBUG and SBUG, and should be valid for several other systems.

The following locations contain the customisable control-codes necessary to implement all the functions of the built-in Line Editor.

HOME $0103.
HOMEs the cursor to the top-left corner of the screen, without clearing the display. This key is also used as an **EXPRESS-LEFT/RIGHT** cursor control. Should be standardised on ^A ($01). $010E. The HOME key on some terminals homes the cursor to the bottom-left of the screen instead of the top-left corner, and so an ESC-sequence has been provided for, which will divert the cursor to its correct position at the top-left. This string should be adapted as appropriate to your terminal, or the first byte ($1B) set to $04 if your terminal already homes correctly.

CLEFT $0104.
Cursor-Left. Moves the cursor left by one character-position without deleting any characters passed over. Currently ^B ($02).

CRIGHT $0105.
Cursor-Right. Moves the cursor right by one character-position without deleting any characters passed over. Currently ^E ($05).

LF $0106.
Line-Feed. Interpreted by the Line-Editor as 'Feed Line back to RBASIC', and returns the line being edited to RBASIC as though it had just been typed in by you. Should be standardised on the current ^J ($0A).

FF $0107.
Clear Screen. Both clears the display and returns the cursor to the top-left of the screen. Should be standardised on ^L ($0C).

REPEAT $0108.
Recalls for editing the Line presently in RBASIC's Input-Buffer, normally the last line typed by you, or returned via LF above. Should be standardised on ^R I($12).

INSCH $0109.
Insert Character. Inserts a SPACE to the left of the present cursor-position, and moves the remaining text right by one character-position, leaving the cursor located over this SPACE. Currently ^T ($14).

DELCH $010A.
Delete Character. Deletes the character directly under the cursor (unless it is the final CR), and closes up the remainder of the text-line. Currently ^U ($15).

SPLIT $010B.
SPLITs a line being edited at a point immediately below the cursor, generally at a ':', passing the text to the left back to RBASIC as a new line, and retaining the text to the right for further editing. Currently ^V ($16).

MELD $010C.
MELD-LINE. Regardless of cursor-position, this key appends to the line presently being edited the next successive line in the RBASIC program, but leaves the added line intact in its original location. Currently ^W ($17).

The simplest way to change the above codes to suit your particular terminal is by way of a Disk-Editing utility, such as DISKEDIT. If not available, then the final address of RBASIC should be ascertained by means of a MAP utility. RBASIC should then be called into memory and exited via the MON command. Using your Monitor's Memory-Examine/Change function, change the codes appropriately and return to FLEX. All that now remains is to save the new version to disk with 'SAVE 0.RBASIC.CMD 0000 xxxx 0000', where 'xxxx' is the final address of RBASIC previously determined.

If you do not know which control-codes have been assigned to the required function-keys on your particular keyboard, they can easily be found with the supplied KEYCODES utility-command. Type 'KEYCODES' in response to the FLEX prompt, and then type each control-key on your keyboard. Hit the DEL key to return to FLEX.

You should also install the supplied ERRORS.SYS on your Systems-Disk, so that the full range of verbal error-messages will be displayed, instead of mere Error-Numbers.

TABLE OF STATEMENTS, FUNCTIONS AND COMMANDS

STATEMENTS	FUNCTIONS	COMMANDS
** ! = INPUT	+,-,*,/,^ or **	+
? = PRINT	ABS	** APPEND
AS	AND	CLEAR
* CHAIN	** ARC	COMPILE
CLOSE	ASC	CONT
DATA	ATN (for compatibility)	** EDIT
DEF	CHR$	* LIST
** DEG	COS	LOAD
DIGITS	CVT%$	MON
DIM	CVT$%	NEW
* DOS (Replaces 'FLEX')	CVTF$	** PDEL
END	CVT$F	RUN
* EXEC	DATE$	SAVE
FIELD	* ERL	SCALE
FOR-NEXT	ERR	
GET	EXP	
GOSUB	FRE	
GOTO	HEX	
IF-THEN-ELSE	* INCH$	
INPUT	INSTR	* RENUM
INPUT LINE	INT	
* KILL	LEFT$	
LET	LEN	
LSET	LOG	
NEW, OLD	MID$	
ON ERROR	** MOD (implied), RESIDUE	
ON GOSUB	** NAME	
ON GOTO	NOT	
OPEN	OR	
POKE, DPOKE	PEEK, DPEEK	
PRINT	PI	
PRINT USING	POS	
PUT	PTR	
READ	RIGHT$	
RECORD	RND	
REM	SGN	
* RENAME	SIN	
RESTORE	SPC	
RESUME	SQR	
RETURN	STR$	
RSET	TAB	
STOP	TAN	
SWAP	USR	
* TRACE	VAL	
	** XOR	

* Expanded or enhanced capability compared to XBASIC
** Additional instruction, not found in XBASIC

ERROR-CODES FOR MICRONICS RBASIC

1	Illegal FMS Function Code
2	Requested File is in use
3	File already exists
4	File could not be found
5	Directory error - Reboot System
6	Directory space full
7	All Disk-space has been used
8	End-of-File error
9	Disk File READ error
10	Disk File WRITE error
11	File or Disk is Write-Protected
12	File is protected
13	Illegal FCB specified
14	Illegal Disk Address
15	Illegal Drive-Number
16	Drive not ready
17	File protected - access denied
18	File Status error
19	FMS Data-Index range error
20	FMS inactive - reboot System
21	Illegal File specification
22	File-Close error
23	Sector-Map overflow
24	Non-existent Record-Number
25	Record-Number match error - file damaged
26	FLEX command error
27	Command not allowed while printing
28	Wrong hardware configuration
30	Data-Type mismatch
31	Out of data in READ
32	Bad argument in ON statement
34	Programmable Control-C (^C) trap
37	FLEX ESC-RETURN trap while printing
40	Bad File-Number
41	File already OPEN
42	Must open File as NEW or OLD
43	File not OPEN
44	File-Status error
45	Field-size error (>252 or <0)
46	Can't extend Sequential File
47	Record 0 not allowed
48	Must use Random type File
50	Unrecognisable Statement
51	Illegal character
52	Syntax Error
53	Illegal Line Termination
54	Line-Number 0 not allowed
55	Unbalanced Parentheses
56	Illegal Function ref.
57	Missing quote
58	Missing THEN in IFstatement
60	Line not found
61	RETURN without GOSUB
62	FOR-NEXT nest error
63	Can't CONTinue
64	Source not present
65	Bad File - won't LOAD
66	RESUME not in ERROR-routine
67	Can't change SCALE-Factor
70	Data-type mismatch in PRINT USING
71	Illegal format in PRINT USING
72	Mixed Mode error
73	Illegal expression
74	Argument <0 or >255
75	Argument >32767
76	Illegal Variable-type
77	Array-reference out of range
78	UnDIMensioned array reference
79	Bad argument in SWAP statement
80	Memory Overflow
81	Array already DIMensioned
83	String too long
90	Undefined USR function
91	Undefined USR call
94	Bad String-Length
95	SIN, COS or TAN must follow ARC
96	Out-of-range in ARC
100	Expression too complex
101	Over/Underflow in FP op
102	Argument too large
103	Division by ZERO
104	Number too large to convert to integer
105	Negative/Zero argument in LOG
106	Conversion error in Integer INPUT
107	Imaginary Square-Root
108	Conversion error (number too large)
109	Over/Underflow in Integer op
255	Illegal Token

REGISTRATION

Full particulars of your Name, Address, etc., have been automatically registered by us, so there will be no need for you to submit a separate Registration Form.

We would appreciate your advising us of any bugs you may come across, though we naturally hope there won't be too many of these little beasts, or that they're too serious. Please supply as much detail as possible, with example program(s), so we can easily re-create and eliminate any such bug. We thank you for your cooperation!

Micronics Research Corp
November 1988