MicroWorld BASIC Interpreter for the 2650

Written by Ian Binnie

Manual by Martin Hood

MicroWorld BASIC Interpreter

for the 2650

Written by Ian Binnie

Manual by Martin Hood

Copyright Notice

Notice to User

Every effort has been made to make this software and its associated manual as accurate and functional as possible. The liability of MicroWorld will be limited to making available to purchasers such updates and corrections as may be found necessary. MicroWorld reserves the right to make alterations to the software supplied at any time without notice.

It is the responsibility of the user to determine the suitablility of these materials for his use.

CONTENTS

SECTION 3

APPENDICES

USING THIS MANUAL

This manual is divided into three main sections.

If you know what you are doing, then go straight to Section 3 which tabulates in summary form, the syntax of this BASIC and other pertinent details related to using the interpreter.

If you know nothing at all, then start at Section 1 and work your way through. If you really know nothing about BASIC, then this manual may not be sufficient on its own. It does not pretend to be a comprehensive "Teach Yourself BASIC" book.

You will do well to skim this manual the first time through and get a vague, overall idea of everything that is in here. Then re-read it in more detail. An attempt has been made to develop the concepts sequentially. However, occasionally it has been neccessary to introduce a topic out of sequence, with little explanation at that point, in order to treat completely the current idea. So although this manual tries to follow a sequential order, it is not always possible to achieve this ideal. If you do not understand a section on first reading, keep reading and trying out the examples yourself, and return to that section later. It will gradually all fall into place.

Section 2 provides a comprehensive explanation of each instruction and function, together with numerous examples to illustrate the uses of each. If you find yourself asking the question "I wonder what would happen if...", don't look for the answer in this manual. Try it on your computer, for that is the ultimate reference. That way you will obtain THE correct answer.

WHAT THIS BASIC IS

This BASIC interpreter for the Signetics 2650 microcomputer, has been written to conform as closely as possible to ANSI standard BASIC. As such it should be possible to enter BASIC programs from various sources and run most of them without modifications being required to accomodate any of the quirks of this particular version of BASIC. Despite this intention, because of the wide variation of versions of BASIC, it is necessary to list a number of instances where this BASIC may differ from the BASIC with which you are familiar. A few of these differences result because this BASIC requires a more structured syntax in some instructions, while others occur because different versions of BASIC execute the same instruction in different ways and one of these options has to be selected from the available choices.

For example a clarification needs to be made about the STRING function RIGHT$(A$,N). Some versions of BASIC, interpret this instruction to mean "return the substring of A$ from the Nth character to the end of A$". This BASIC, in common with most microcomputer versions of BASIC, will return the "last N characters of the string A$".

The second consideration concerns the naming of VARIABLES. NUMERIC VARIABLE names can be either a single letter, or a single letter followed by a single digit between 0 and 7 inlusive. However 8 and 9 are not permitted. A significant simplification of the routines handling the variables occurs by calculating in base 8 rather than base 10. It is thought that this will not be an excessive limitation since many versions of BASIC do not allow two character names at all.

Thirdly, KEYWORDS must not contain imbedded spaces. GO TO and GO SUB containing a space are NOT permitted and will be diagnosed as a SYNTAX ERROR.

The fourth restriction this BASIC imposes concerns the DIMensioning of ARRAYS. All ARRAYS must be DIMensioned. Implicit dimensioning of arrays is not provided. Grouping DIMension statements at the start of a program, together with comments to explain the purpose of each VARIABLE, is good programming practice.

Fifthly, both of the functions LOG(X) and EXP(X) are calculated to base 10 rather than base e.

Finally only ONE DIMENSIONAL ARRAYS are supported by this version of BASIC. A two dimensional array (or a multi-dimensional array) can be synthesised quite easily by appropriate programming techniques, so this is not a severe

limitation. This technique is described in the section about VARIABLES.

If you are new to BASIC programming, spend a lot of time trying out the numerous examples illustrating each of the instructions. Invent examples of your own. If you are an experienced BASIC programmer, you should enjoy the experience of using this reasonably complete, standard version of BASIC on your 2650 microcomputer system.

SYSTEM REQUIREMENTS

Minimum requirements of your system to run this BASIC interpreter are:-

1. A monitor program located at $0 to $3FF such as PIPBUG(1) or BINBUG(2). If another monitor is used then it will be necessary to change the I/O addresses. These are detailed in appendix e.

2. Read/Write memory between $500 and $7FF for Interpreter scratchpad.

3. Either Read Only Memory containing the BASIC interpreter, or Read/Write Memory between $800 and $1FFF to store the BASIC interpreter program.

4. Read/Write memory commencing at $2000 for source program and array variable storage. The size of the program which can be run will depend on the amount of memory available here. 4K is a recommended minimum. The start address of program storage may be changed by altering the address constant in $1477 - $1478 to point to the new storage starting address. Refer to appendices d and e for more details.

START ADDRESS

The START ADDRESS for the BASIC interpreter is $800. Typing G800, if using PIPBUG or BINBUG, will commence execution of the BASIC interpreter.

For specific details about adapting this BASIC to your system, refer to the attatchment describing the 'Personality module' installed in your BASIC.

(1) PIPBUG copyright SIGNETICS. (2) BINBUG copyright I.Binnie.

CONSTANTS, VARIABLES AND EXPRESSIONS

A clear understanding of each of these three fundamental concepts is required before you will be able to understand much of the rest of this manual. This is because these terms will be used freely and it will be assumed that the reader knows the meaning of the terms CONSTANT, VARIABLE and EXPRESSION as applied to BASIC.

CONSTANTS

A CONSTANT is a piece of data contained in a BASIC program which is fixed and does not change with the running of the program. Two types of CONSTANT are allowed. A CONSTANT may be a NUMERIC CONSTANT, in which case it will be a number.

For example:- 6 284.3 1.98E-6 -99.2

Alternatively, a CONSTANT may be a STRING CONSTANT, in which case it will consist of a string of any printable characters.

for example:- "FRED" "HELLO THERE"

A STRING CONSTANT is normally enclosed in quotes to ensure that the BASIC interpreter knows where it begins and ends.

Each of the lines of BASIC below contains a CONSTANT.

```
10 LET A=1.46                1.46 is a NUMERIC CONSTANT
20 PRINT"HELLO"              "HELLO" is a STRING CONSTANT
```

VARIABLES

A VARIABLE is a group of memory locations which BASIC uses to store either a number or a string of characters. Each group of memory locations is referenced by a NAME. The following are some examples of valid variable names. See page 20 for more details about naming variables.

```
A    A1   D    D7        NUMERIC VARIABLE NAMES
A$   G$                  STRING VARIABLE NAMES
```

A group of memory locations used to store a number is called a NUMERIC VARIABLE, while a group of memory locations used to store a string of characters is called a STRING VARIABLE. The main difference between CONSTANTS and VARIABLES is that the value of a CONSTANT is fixed, whereas the value or contents of a VARIABLE may be altered during the running of a program.

EXPRESSIONS

An EXPRESSION is any valid combination of the following.

CONSTANTS
VARIABLES
MATHEMATICAL OPERATORS
FUNCTIONS

Examples of EXPRESSIONS

```
(A+4)*B
INT(RND(1)*100)
LEFT(A$,A-8*4)
VAL(MID(N$,4,3)
```

BASIC will nearly always allow an EXPRESSION to be used in any place in a statement where a single variable is specified.

For example:-

```
LEN(A$+"FRED"+B$)
SIN(360*E+0.1)
RIGHT$(A$+B$,B+C/3)
```

BASIC will evaluate any EXPRESSIONS first, according to its priority rules. These are treated in detail in the section on mathematical operators. If brackets are included to specify an order of evaluation for an EXPRESSION then BASIC will commence at the innermost brackets and work outwards.

For an EXPRESSION to be valid, each of its components must evaluate to either a NUMBER or a STRING. It does not make sense to try to evaluate an expression which consists of a mixture of numbers and strings. Both of the following EXPRESSIONS are invalid because they attempt to mix components which evaluate to both numbers and strings in the one EXPRESSION. Both will generate a TYPE ERROR.

```
10 LET A$=B$+A          INVALID
20 A=SIN(A$+1)          INVALID
```

HOW COMPLICATED CAN AN EXPRESSION BE ?

The length of one line of BASIC code is limited to 79 characters, the length of the input buffer, so an expression can never be longer than one line. A BUFFER ERROR will result if more than 79 characters are entered on the one line.

More importantly, the intent of a long expression is usually obscured by its complexity. It is better programming practice to break a long expression up into a couple of short expressions. Short lines are also easier to correct when you make a typing error. The following long expression solves the resistance of three resistors in parallel.

```
5 REM R = EFFECTIVE RESISTANCE OF R1, R2, R3 IN PARALLEL
10 R=1/((1/R1)+(1/R2)+(1/R3))
```

This can be much more clearly written as four lines.

```
5 REM C = CONDUCTANCE
10 C1=1/R1
20 C2=1/R2
30 C3=1/R3
40 C=C1+C2+C3
50 R=1/C
```

Although this second example does use more VARIABLE NAMES and hence a little more memory, the meaning of the expression is much clearer.

Remember that BASIC will allow a NUMERIC EXPRESSION anywhere a NUMERIC VARIABLE is specified, and will allow a STRING EXPRESSION almost anywhere a STRING VARIABLE is specified.

ENTERING A PROGRAM

The first three COMMANDS with which to become familiar are

NEW OLD LIST

NEW tells BASIC that it should delete from memory all traces of any previous program and that you are about to enter a new program.

OLD tells BASIC that there is already a BASIC program in memory which you have previously typed in or loaded from tape, and that it should use this.

You MUST do either a NEW or an OLD or a LOAD before doing anything else. If you don't 'open a file' then you will get a FILE ERROR. If you do an OLD and no valid file exists, you will also get a FILE ERROR.

Both NEW and OLD can be abbreviated to their first letter only. Typing 'N' instead of 'NEW' of 'O' instead of OLD is perfectly acceptable.

LIST will list the program you have entered. LISTing is discussed in greater detail later. For the moment it is sufficient to know that typing 'L' will list your entire program. The LIST command is discussed in greater detail on page 14.

YOUR FIRST PROGRAM

Suppose this is your first time and there is no OLD BASIC program in memory. After typing NEW (or just N), type a Carriage Return, <cr>. Every time you want BASIC to act upon the line of text or commands you have just typed in, press the Carriage Return key <cr>. Nothing will happen until you do. If you make a mistake while typing in a line you can use the Backspace key (Control and H pressed at the same time if your keyboard doesn't have a special Backspace key) to backspace along the line to the mistake, then simply type the correct letters over the top. If you really make a mess of typing a line, you can delete the whole line by pressing the DELete key. This will delete the whole line and you can start all over again. The line will end up looking like this

```
-CURSES MY TPYNDG ID DYSLECTIC***
-
```

The three asterisks tell you that the whole line has been deleted.

A SAMPLE PROGRAM

Try entering the short program below. At the moment you may not know what each of the BASIC statements does. This does not matter for now. Getting the program into the computer is the important thing. Just type in each line as it appears. Do not forget to press the CARRIAGE RETURN, abbreviated <cr> elsewhere in this manual, at the end of each line.

```
10 PRINT "WHAT IS YOUR NAME";
20 INPUT N$
30 PRINT "HELLO ";
40 PRINT N$
50 END
```

RUNNING THE SAMPLE PROGRAM

Once the program has been entered and any mistakes have been corrected by retyping the lines, you can command BASIC to start executing the program by typing

RUN <cr>

or the abbreviation 'R'. BASIC will start executing the instructions contained in the program beginning at the lowest line number, in this case 10, and working up.

WHAT WILL HAPPEN

Since BASIC makes use of everyday words in the English language, you have probably guessed some of the things this program will do already. First the computer will print

WHAT IS YOUR NAME?

on the terminal. Line 10 causes 'WHAT IS YOUR NAME' to be printed and then the INPUT statement on line 20 prints the question mark to prompt you to input data from the keyboard. When you have typed in your name, and after you press <cr> to tell BASIC that you have finished typing it, (how else is it going to know ?) BASIC takes the string of characters in your name, and assigns it to the string variable called N$.

Line 40 causes BASIC to print the string constant "HELLO". It is called a CONSTANT because it doesn't change. BASIC knows that it is a string because it enclosed in quotes. The semicolon at the end of the line tells BASIC to NOT start a new line at the end of the print statement, otherwise it would automatically do so.

The reason for inhibiting the new line is because line 50 prints out your name, which is stored in the variable N$, and this should appear on the same line as "HELLO".

LINE NUMBERS

Some things are immediately obvious even in this short BASIC program.

1. Every line begins with a number.

   Line numbers serve several purposes in BASIC. They tell the BASIC interpreter that " the line being input is a line to be stored in memory" rather than acted upon immediately. If the line does NOT begin with a number then the BASIC interpreter assumes that the line is a command like NEW or OLD or some other instruction and that you wish it to act on that instruction immediately. Most BASIC instructions can be executed directly. This is called IMMEDIATE MODE OPERATION and is described in more detail later.

   The second purpose of a line number is to identify the order in which BASIC is to execute the instructions. BASIC will start at the lowest line number and proceed upwards, unless a branch instruction in the program interupts this sequence. If the program does contain branch instructions, then the line numbers provide a destination address for the branch to go to.

   Thirdly if you want to alter the program, you can change a line by simply typing the new line using the same line number as the line you want it to replace. Alternatively you can insert a new line into the program by giving it a line number between two existing line numbers. That is why the line numbers of the sample program increment by 10 each time, to allow plenty of space for extra lines to be added. The maximum line number allowed is 9999, so there are ample numbers available for even the longest program.

2. The first word in each line is an INSTRUCTION to BASIC to do something.

   This is called a KEYWORD. A KEYWORD is analogous to the verb in an English sentence and a STATEMENT is analagous to the complete sentence. Each line can contain more than one STATEMENT if each statement is separated by a colon ':'.

MULTIPLE STATEMENT LINES

The sample program above could have been written in a much more concise form by including more than one BASIC statement on each line.

```
5 REM A TEST PROGRAM TO SAY HELLO TO YOU
10 PRINT"WHAT IS YOUR NAME"; : INPUT N$ : PRINT "HELLO ";N$
20 END
```

A MULTIPLE STATEMENT LINE consists of BASIC statements typed on the same line and separated by a colon, ':'. The maximum length of a MULTIPLE STATEMENT LINE is 79 characters, since this is the length of the input buffer. A BUFFER ERROR will result if more than 79 characters are entered on the one line.

Normally it will not make any difference to the execution of a program whether the lines are typed in separately or typed in with multiple statements on the one line. However if the line contains an IF - THEN statement, the statements on the line after the THEN will only be executed if the IF - THEN statement is TRUE. If it is FALSE, the program will not execute any of the statements following the THEN, but will continue to the next numbered line of program. Often this feature can be used to advantage to avoid using a GOTO instruction since it allows more than one instruction to be executed if a test is TRUE.

MULTIPLE STATEMENT LINES are not a feature that should be used indiscriminantly. There is considerably more work involved in correcting a MULTIPLE STATEMENT LINE than in retyping one of the three or four single lines which would be its equivalent.

Using a MULTIPLE STATEMENT LINE does save the small amount of extra memory space which a line number uses, but unless you are a very confident programmer or the additional statements form part of an IF - THEN statement, it is probably better to use a single statement per line, at least initially.

LISTING

To LIST the program you have entered use the LIST command. The abbreviation 'L' will also work. Typing LIST <cr> will list the entire program. If your program is very long it is probable that you will only want to look at a small section at a time. The command

```
LIST 20-30
```

will list all the lines between line 20 and line 30, in the case of the example on page 12

```
20 INPUT N$
30 PRINT "HELLO ";
>
```

will be output to the terminal. Even if the lines specified don't actually exist, the LIST command will print out all the lines with numbers between those specified.

```
LIST 20
```

will list only line 20. The list command could be abbreviated to

```
L20
```

and still produce the same results as before.

The output of a LISTing can be directed to one of the other output ports by using a statement of the form

```
LIST#n,20-30
```

where n is the number of one of the I/O ports. See appendix f for more details.

Pressing the BREAK key during listing will stop the listing and return BASIC to the edit mode.

END

Line 60 is an END statement. It tells BASIC that this is the end of the program. The END statement should be the last statement in the program.

REM

A line beginning with REM is considered by BASIC to be a comment or remark line. Anything after a REM on a line is skipped over by BASIC, including any multiple statements! REM is used to insert comments into a program to provide an explanation of what is meant to be happening. REMarks do take up memory space and also take time to execute. A REMark may not have any other statements following it since BASIC will ignore them since it considers everything which follows the REM to be a comment including colons. A REMark may not follow a DATA statement on the same line or it will also be interpreted as data.

THE IMMEDIATE MODE

One of the most helpful features of BASIC as a high level language is the interactive feature known as the IMMEDIATE MODE. BASIC allows you to type in a line of instructions and execute it immediately by pressing return. To execute a BASIC statement in the IMMEDIATE MODE, merely type in the statement without a line number. For example:-

```
PRINT"THIS IS AN IMMEDIATE MODE STATEMENT"
```

will cause

```
THIS IS AN IMMEDIATE MODE STATEMENT
```

to be printed on the terminal.

A more useful example is illustrated below. The BASIC line

```
FOR I=1 TO 10 :PRINT I,I*I,I*I*I :NEXT I
```

will cause the following table of squares and cubes of the numbers between 1 and 10 to be printed as follows.

```
1         1         1
2         4         8
3         9         27
4         16        64
5         25        125
6         36        216
7         49        343
8         64        512
9         81        729
10        100       1000
```

All this can be done with one line of BASIC code. The IMMEDIATE MODE allows your computer to be used as a powerful calculator merely by typing in a line of BASIC instructions.

Some instructions are not permitted in the IMMEDIATE MODE. For example, DATA, INPUT, STOP, and RETURN make no sense when executed in the IMMEDIATE MODE. Each of these instructions will be discussed in greater detail later in this manual.

Once the execution of a program has been halted, either by a STOP statement or by pressing the break key, the current values of each of the variables may be inspected using the IMMEDIATE MODE. So by appropriately placing STOP statements and CONTinuing, it is possible to step through the program and see exactly what happens to each variable at every statement. For example typing

PRINT A <cr>

without a line number, will cause the current value of the variable A to be printed out. The IMMEDIATE MODE is an ideal method of 'trying out' BASIC instructions to see what they do, particularly if you are unfamiliar with BASIC. Try some and see.

STOP - CONTINUE

STOP does precisely what it says - it stops the execution of the program. STOP leaves everything 'as is', all variables will still contain their current value and all arrays will remain set up so that the contents of these may be inspected using PRINT statements in the IMMEDIATE MODE. After the contents of the various variables have been inspected, and provided no changes have been made to the program, the execution of the program can be resumed from where it stopped by using the CONTINUE instruction to continue.

END, however, signifies the physical end of the program as well as terminating the execution of the program.

Suppose the program listed below was printing out unexpected values. You could insert a STOP instruction at line 55, examine the values of C, C1, C2 and C3 and then continue with the execution of the program by using CONTINUE.

```
10 REM TEST PROGRAM TO DEMONSTRATE THE STOP INSTRUCTION
20 C1=1/R1
30 C2=1/R2
40 C3=1/R3
50 C=C1+C2+C3
60 R=1/C
70 END
```

The instruction

```
55 PRINT"STOP AT 55" : STOP
```

inserted into the program, would cause BASIC to stop executing the program at line 55 and print

```
STOP AT 55
```

Suppose that 1 had been entered as R1, 2 as R2 and 4 as R3. The value of the variables C1, C2, C3 and C could be inspected by typing

```
PRINT C1,C2,C3,C
```

which would cause the following to be printed out.

```
1        .5        .25       1.75
```

Other STOP instructions could have been inserted at other places in the program. When execution of the program was resumed using CONTINUE, it would have stopped again at the next STOP statement. CONTINUE cannot be used if you have modified the program, because when you specify the line number of the line you wish to replace, BASIC loses its pointer to the line number at which it is to continue, as well as destroying a number of other pointers.

SIZE

The SIZE command will print on the terminal the number of memory locations used to store the program, in decimal. Also if the program has just been run, the number of memory locations used for both the program and variable storage will be printed as well.

In fact the number printed will be one less than the length. This is because of the format of the file which begins with a $Ø2, STX marker followed by a <cr> and terminates with a <cr> followed by a $Ø3 ETX marker. Because the three bytes (Ø2)<cr>(Ø3) are present even when no program is present, the SIZE command immediately after a NEW will return a size of 2, the length of the file minus 1.

SAVE

The SAVE command can be used to save the BASIC source file currently in memory. SAVE will cause the entire source file to be output in a standard format as ASCII characters, exactly as it is stored in memory. The dump is preceeded by a leader of 32 DEL characters ($FF), and is followed by a trailer of NULLs ($ØØ), 254 for a 3ØØ baud dump and 8Ø for a 11Ø baud dump. The NULLS ensure that you have time to turn off the cassette recorder before the next recording starts, and also will be interpreted by BASIC as a BREAK, so if it misses the (Ø3) ETX it will exit from the load on the trailer.

The format of the source file dumped is as follows:-

```
32 x DEL (Ø2)<cr> text <cr>.....<cr> text <cr>(Ø3) NULLs
```

This dump format is compatible with the SAVE routines of many other versions of BASIC. The actual output routine used to output the characters will depend on the particular "personality module" installed in your version of BASIC, and can be changed to suit particular requirements.

LOAD

The LOAD command will load a SAVEd program back into memory. It expects the program to be input in the same format as it was dumped using the SAVE command. The LOAD routine looks for the (Ø2) STX as its start character and a (Ø3) ETX or a NULL as the end character. No error checking is performed by the loader. The input routine used by the loader will also depend on the 'personality module' installed in your particular version of BASIC. For specific details, refer to the refer to the attatchment describing your 'personality module'.

VARIABLES

WHAT IS A VARIABLE

A VARIABLE is a temporary storage area for a piece of DATA which is used by the BASIC interpreter. Each VARIABLE or group of VARIABLES has a unique name so that it can be referenced by a BASIC program. As its name suggests a VARIABLE can be changed by the program. This BASIC allows two types of VARIABLES, NUMERIC VARIABLES and STRING VARIABLES and these may be used either alone or as an ARRAY.

TYPES OF VARIABLE

A NUMERIC VARIABLE stores a NUMBER. The number must be within the range plus or minus 1.00000E-64 to plus or minus 9.99999E 62, where E-64 means "multiplied by 10 raised to the -64th power" and E 62 means "multiplied by 10 raised to the 62nd power". NUMERIC VARIABLES are stored in Floating Point Binary Coded Decimal format to avoid the rounding errors inherent in Binary storage format and are accurate to 6 significant figures.

A STRING VARIABLE stores a string of characters and treats them as one entity. An example of a STRING would be a line of this text, or your name, in fact any group of alphanumeric characters. A STRING VARIABLE can be a maximum of 158 characters long in this BASIC.

VARIABLE NAMES

The type of a VARIABLE is indicated by its name.

A NUMERIC VARIABLE name consists of either a single letter or a single letter followed by a digit between 0 and 7 inclusive. In this respect this BASIC differs from others which allow the use of 8 and 9 as well. However many versions of BASIC do not allow two character variable names at all and provided this limitation is remembered, it should not cause any difficulties.

A STRING VARIABLE name consists of a single letter A to Z, followed by '$'. For example, 'J$' is a valid name for a STRING VARIABLE.

All the following are examples of valid variable names.

A B4 G$

These are NOT valid variable names.

B9 The number in the name is not between 0 and 7
SC More than one letter in the name
G3$ Numbers are not allowed in STRING VARIABLE names

ARRAY VARIABLES

Often it is useful to group VARIABLES together into an ARRAY. An ARRAY allows you to group variables together using the same variable name, but be able to access each variable individually by means of an index. This BASIC allows both SINGLE DIMENSIONED NUMERIC ARRAYS and SINGLE DIMENSIONED STRING ARRAYS.

NAMING ARRAY VARIABLES

An ARRAY VARIABLE name consists of a single letter only (no number is allowed) or a single letter followed by a '$' in the case of a STRING VARIABLE, followed by an index variable in brackets. For example, A(N), where A is the name of the ARRAY, and N is the index to the particular entry in that ARRAY.

WHAT IS AN ARRAY ?

An array can be visualised as a row of houses in a street. The street name (variable name) is the same for all the houses, but you indicate a particular house by specifying its number (index). One use for a STRING ARRAY might be to set up a table of owner's names vs. house number. The array could be called Q$(4) and might look like this:-

INDEX	CONTENTS
Q$(1)	BINNIE
Q$(2)	HILL
Q$(3)	SMITH
Q$(4)	JONES

To print the name of the owner of the second house you would access the second element in the array using a statement such as

```
PRINT Q$(2)
```

Here 'HILL' would be printed out. The index in brackets indicates which entry in the table of variables in the array is to be accessed.

DIMENSIONING ARRAYS

Since ARRAYS of variables can consume a considerable quantity of memory, or only a small amount if only a few variables are involved, it is unreasonable to expect BASIC to guess in advance how much memory it will need to allocate to store a given ARRAY. BASIC is told how many variables to reserve for an array by using a DIMension statement. In this BASIC all arrays must be DIMensioned. Many BASICs will allow 'implicit' DIMensioning of an ARRAY and will automatically assign an ARRAY of ten VARIABLES when the ARRAY VARIABLE name is first used in the program. This can be very wasteful of memory if only a small ARRAY is needed and it is better programming practice to explicitly DIMension each array by using a statement such as

```
DIM A(3)
```

This will reserve memory space for an ARRAY of FOUR VARIABLES, A(Ø), A(1), A(2), and A(3). This array will actually contain one more variable than is specified in the DIM statement. Unlike many other microcomputer versions of BASIC, but in accordance with the ANSI standard, this BASIC provides an element with an index of zero in each array. The zero element is a useful place to store information about the number of elements of the array which have been used.

A feature not permitted by many other BASICs, allows the DIMension statement to contain a variable. Thus

```
DIM A$(N)
```

is permitted, provided 'N' has been previously assigned a non-zero value. This can be particularly valuable when the size of the array required by a program depends on the circumstances in which the program is being run. Where only constants are permitted in a DIMension statement, it is necessary to estimate the maximum size of the array ever likely to be required and probably allocate much more space than is normally used. If the program first determines the likely array space required, an array of appropriate size can be dimensioned.

MAXIMUM SIZE OF ARRAYS

A numeric array may be DIMensioned to any size desired, provided this does not exceed the available read/write memory. If an array is DIMensioned which exceeds the available memory, a MEMORY ERROR will result. However a string array may only contain a maximum of 255 elements. An attempt to DIMension a string array greater than 255 elements in length will not produce an error, but should not be attempted because it may produce unpredictable results.

REDIMENSIONING ARRAYS

An array may be dimensioned only once during the course of any program. A DIM ERROR will result if an attempt is made to DIMension an array with the same variable name, even if it contains a different number of elements, more than once in a program.

DOUBLE DIMENSIONED ARRAYS

Although DOUBLE DIMENSIONED ARRAYS are not supported in this version of BASIC, they may be quite simply implemented by appropriate programming. For example a 5 x 5 ARRAY can be DIMensioned by

```
DIM A(4*5+4)
```

which will actually reserve an ARRAY of 25 VARIABLES (do not forget the zero element). This can be treated as though it were a two dimensional array by accessing each variable with a statement of the form

```
A(M*5+N)
```

which will retrieve the (M,N)th entry in the array.

Multi-dimensioned arrays may also be synthesised by an extension of this process.

DIMENSIONING ERRORS

A DIM ERROR will result if

1. An attempt is made to assign a value to an element of a NUMERIC ARRAY which
   a. has not been dimensioned, or
   b. would fall outside of the numeric array storage area.
2. Any reference is made to an element of a STRING ARRAY which
   a. has not been dimensioned, or
   b. exceeds the reserved dimension.

eg. DIM A$(20) : PRINT A$(25) will cause a DIM ERROR.

ASSIGNMENT STATEMENT - LET

The instruction LET, causes a variable to be replaced by the value of an expression.

```
10 LET A=B
20 LET A=6.456
30 LET A$=B$
40 LET A$="HELLO THERE"
50 LET A$=CHR$(32)
```

are all valid assignment statements. The LET statement is optional and if not present is assumed to be 'implied'. Thus

```
10 A=B
20 A=6.456
30 A$=B$
40 A$="HELLO THERE"
50 A$=CHR$(32)
```

will each be interpreted by BASIC in exactly the same way as those in the previous list which do have LET specified. The one exception where LET must be specified occurs when assigning a variable in the IMMEDIATE MODE. Here LET MUST be explicitly stated.

```
LET A$="HELLO"
```

is valid in IMMEDIATE MODE, but

```
A$="HELLO"
```

is NOT valid and will cause a SYNTAX ERROR to be generated.

VARIABLE TYPES

BASIC recognises two types of variables, NUMERIC and STRING. It clearly does not make any sense to try to make the numeric variable A contain the string "HELLO THERE". Only variables of the same type can be assigned to each other. So

```
10 LET A=B$                    INVALID
20 LET A="HELLO THERE          INVALID
30 A$=A                        INVALID
40 A$=6.456                    INVALID
```

are all INVALID variable assignment statements because they attempt to assign data which is not of the same type as the variable. In each case a TYPE ERROR message would result.

I/O STATEMENTS

The next group of instructions to be discussed are those which enter data into the program and those which output data from the program to the terminal or another device. As a group these are called INPUT/OUTPUT INSTRUCTIONS or I/O INSTRUCTIONS.

INPUT INSTRUCTIONS

READ and DATA

The fundamental input instruction in BASIC is READ. When BASIC executes the first READ instruction in a program, it gets the first piece of data in the first DATA statement. An example of a READ instruction is

```
10 READ A$,C,D
```

and the form of a DATA statement is

```
20 DATA FRED JONES,1.04,345
```

The READ statement causes one piece of DATA to be read from the DATA statements into each of the variables listed after it. Each of the variables listed in the READ statement must be separated by a COMMA. Similarly each of the pieces of data listed in a single DATA statement must also be separated by a comma. A DATA statement cannot be followed by another statement on the same line, because the following statement including the colon will be interpreted as data.

The READ instruction in the example above will read the string "FRED JONES" from the DATA statement and assign it to the string variable A$, read 1.04 and assign it to the numeric variable C and read 345 and assign it to the numeric variable D. The DATA statement may be placed anywhere in the program, but your programs will run more quickly if they are all placed together at the start. This is because BASIC searches through the program to find the next data item, and the further it has to search, the longer it will take to find it. BASIC will step through the data in the first DATA statement until none is left. Then it will continue on to the next DATA statement, and so on. If a program instructs BASIC to read data when there is no more data to be read, then a DATA ERROR will be generated.

The READ and DATA statements above could have been replaced by the following LET statements and would have produced exactly the same result.

```
10 LET A$="FRED JONES"
20 LET C=1.04
30 LET D=345
```

As is the case with the LET instruction, or any BASIC verb which causes data to be assigned to a variable, the data in the DATA statement must be of the same TYPE as the variable to which it is being assigned. The following READ and DATA statements will cause a TYPE ERROR, because B is a numeric variable and "FRED" is a string.

```
10 READ B
20 DATA FRED
```

Normally in BASIC a string constant must be enclosed in quotes. However in a DATA statement the quotes are optional. If the string data IS enclosed in quotes then the string variable will be precisely what is between the quotes, including any leading spaces. If the data is not enclosed in quotes then any leading spaces will be removed before it is assigned to the variable. So

10 DATA "FRED" is equivalent to 10 DATA FRED

RESTORE

Each time a READ instruction assigns a value to a variable, it moves an internal DATA pointer along to the next piece of data in the DATA statement, or to the next DATA statement if the current line of DATA is exhausted. This pointer can be set back to the first piece of DATA by using the RESTORE instruction. The following

```
10 DATA FRED,1.04,345
20 READ A$,C
30 RESTORE
40 READ Z$,F,G
```

will cause the following values to be assigned to the variables.

A$ will be "FRED" C will be 1.04

Z$ will be "FRED" F will be 1.04 G will be 345

INPUT

Because BASIC is an interactive language, programs can be designed which require an immediate response from the user to determine the subsequent execution of the program. It would be a very dull game of SPACE WAR indeed that required all the commands to be written as DATA statements and included in the program. The INPUT instruction causes the execution of the program to pause. A question mark is output to tell the user that BASIC is expecting some data to be input, and then the data, or data items separated by commas, is entered and the data assigned to the variable or variables. The form of an INPUT statement is

```
20 INPUT#n,<list>
```

where <list> is a list of variable names separated by commas and 'n' is an integer, or an expression which evaluates to an integer, between 0 and 3 and indicates one of the four I/O channels from which the data is to be entered. (see appendix f.)

The INPUT instruction will output a question mark to the I/O channel specified, and then wait for data to be input from that same I/O channel. When the data being input is terminated by a carriage return <cr>, the data just input will be assigned to the specified variable. If insufficient pieces of data, separated by commas, are entered before the <cr> then BASIC will keep prompting with question marks until sufficient pieces of data have been entered.

INPUTTING A NULL STRING

A NULL STRING is a string containing no characters. Entering only a carriage return, <cr>, in response to an INPUT statement will result in a null string being assigned to the variable. A null string, signified as "", without a space between the quotes, is analogous to including a numeric variable containing zero. All string variables are initialised to null strings by BASIC when an OLD or RUN command is executed.

A null string may be tested and compared just like any other string.

```
10 INPUT A$
20 IF A$="" THEN PRINT"A NULL STRING WAS ENTERED"
```

will print

```
A NULL STRING WAS ENTERED
```

if only a <cr> is input from the keyboard.

INPUTTING COMMAS

BASIC interprets a comma as a delimiter between pieces of data, so if the line typed in response to an INPUT statement contains a comma, BASIC will assume that what follows is part of the next piece of data and will ignore it. Entering

```
HOOD,Martin
```

in response to the statement

```
30 INPUT A$
```

will result in A$ being "HOOD" because everything after the comma is presumed to be part of the next piece of data. But if the data being input is enclosed in quotes, BASIC will treat all the data between the quotes as valid input data. Hence

"HOOD,Martin" will cause A$ to be "HOOD,Martin"

MATCHING DATA TYPES

Since the INPUT instruction assigns data to a variable, the type of data being input must match the type of the variable to which it is being assigned. If the variable specified in the INPUT statement is a numeric variable, then ONLY a number can be entered. If any characters are entered which do not form part of a valid number, or a simple numeric expression, then BASIC will generate a TYPE ERROR message. A valid number can contain the integers 0 to 9, and , -, +, ., and E. The following are examples of valid numbers.

```
9675      1.345E-5      -8.3456E+34    123*5+4
```

INPUTTING NUMBERS

When a string variable is specified as the variable in an INPUT instruction, ANY printing characters may be input, since a string may contain any characters including numbers and punctuation.

Actually it is often useful to input a number as a string and after testing it as a string of characters to ensure that it is a valid number, convert the string variable into a numeric variable using the VAL function (see page 53). If a program is being used by a naive operator, it is quite likely that the wrong type of data will be input in response to an INPUT instruction. For example, typing "four" instead of '4' would be a quite understandable mistake. If the statement

```
100 INPUT A
```

were used to input the number, then BASIC would generate a TYPE ERROR message when "FOUR" was entered and worse still would stop the execution of the program and return to the editor mode. Next when our naive operator realises his mistake and eventually types

```
4<cr>
```

line 4 in the program, if one exists, is deleted and the program probably will not work anymore.

The following program will input a number as a string variable and check that only a valid number has been input before converting the string to a numeric variable. This routine only checks a single digit.

```
90 REM A ROUTINE TO CHECK FOR VALID NUMERIC INPUT
100 INPUT A$
120 IF ASC(A$)<48 THEN GOTO 150
130 IF ASC(A$)>57 THEN GOTO 150
140 LET A=VAL(A$)
145 GOTO 200
150 PRINT "NUMBERS ONLY PLEASE"
160 GOTO 100
200 ....program continues....
```

INPUTTING MORE THAN ONE VARIABLE AT A TIME

The INPUT statement is not limited to inputting only one variable at a time. The line :-

```
50 INPUT A$,B,C,D
```

is also valid. Here the user would type in the various pieces of data separated by commas. If insufficient pieces of data are entered on one line, then BASIC will keep prompting with a question mark until sufficient data items have been entered. For the case above, if you typed in

```
FRED,10,45 <cr>
```

then there would be data sufficient for only three of the four variables and BASIC will prompt for another piece of data by outputting another question mark to the terminal.

The types of data in an INPUT statement may be mixed, that is both string and numeric variables may be entered in by the one INPUT statement. However, the correct type of data must be entered for each variable specified. The more variables that are input at the one time in a single INPUT statement, without the

program giving any hints as to whether it expects a number or a string, the more likely the user is to enter the wrong type of data.

If a null string is entered when more than one string variable is expected, then all the variables specified will be set to a null string.

PRINTING A MESSAGE IN AN INPUT STATEMENT

Usually more prompting is required than merely a question mark to elicit the appropriate response to an INPUT statement. BASIC allows a message to be printed to the terminal as well as the question mark. The statement

```
20 INPUT"WHAT IS YOUR NAME ";N$
```

will cause

```
WHAT IS YOUR NAME ?
```

to be printed on the terminal and your response to be assigned to the string variable N$. The semicolon is required as a delimiter between the string to be printed (in quotes) and the variable to be input. Either a comma or a semicolon will work in this position, but one of these two delimiters must be present. The maximum length of the message which can be printed by this method is limited by the maximum length of one line of program which is 79 characters. If the requirement arises to print a longer message, this can be done using a separate PRINT statement for the message, followed by an INPUT statement to input the data.

ABORTING FROM AN INPUT STATEMENT

A special case occurs when it is required to abort a program while it is waiting for input. Merely pressing the BREAK key would not be adequate since BASIC cannot distinguish between this and a string of nulls. To abort from an input statement the ESCape key is used. Pressing the ESCape key while the program is waiting for data to be input will terminate the execution of the program and return BASIC to the editor mode. At this point the IMMMEDIATE mode can be used to inspect any variables, or the program may be modified. At any other time during the execution of a program, pressing the break key, or indeed any key, will halt the execution of the program and return you to the edit mode.

OUTPUT INSTRUCTIONS

PRINT

The general output statement in BASIC is PRINT. This statement causes the data specified in the PRINT statement to be output to the stated output channel and hence to the device connected to that channel.

The form of a PRINT statement is

```
10 PRINT#n,<list>
```

where <list> can be any list of variables, constants, or expressions either string or numeric, separated from each other by a comma ',' or a semicolon ';'. #n is the number of the output channel and may be a numeric expression. There are four output channels numbered 0 to 3. If the output channel is not specified, then the default value of 0 is used, the user's terminal. Thus

```
10 PRINT A$    is equivalent to    10 PRINT#0,A$
```

For more information about I/O channels, refer to appendix f.

PR is a valid abbreviation for PRINT, and

```
10 PR "HELLO"  is equivalent to    10 PRINT "HELLO"
```

PRINT DELIMITERS

The use of a semicolon, ';', to separate the items in a PRINT statement causes the next item in the print list to be printed immediately adjacent to the previous one.

```
10 LET A$="###"
20 PRINT#0,"***";A$;"$$$"
```

will print

```
***###$$$
```

PRINT ZONES

Alternatively, using a comma to separate the items in a print statement, will cause the first character of the next item to be printed in the first position of the next 'print zone'. The

display is divided into a number of 'print zones', each 13 characters wide. If all three semicolons in the previous example were replaced by commas, the following would have been printed instead.

```
***           ###           $$$
```

This simple formatting is very useful for separating numbers out into columns to make them more readable. A semicolon after the last item in a PRINT list will inhibit the normal carriage return and linefeed generated at the conclusion of a print statement and cause the output from the next print statement to be printed on the same line. For example

```
10 LET A=1.004
20 PRINT "A =";
30 PRINT A
```

will be printed as

```
A = 1.004
```

EXPRESSIONS IN PRINT STATEMENTS

A PRINT list can also contain any resolvable BASIC expression. For example

```
10 PRINT"AREA OF CIRCLE =";3.1416*R*R;" SQUARE UNITS"
```

is a valid PRINT statement. The arithmetic expression will be evaluated and the resultant value printed out along with the string constants. If R was 10 then this would be printed.

```
AREA OF CIRCLE = 314.16 SQUARE UNITS
```

PRINTING NUMBERS

Numbers greater than 0.1, but less than 999999, will have the first significant six figures printed in Floating Point format with any trailing zeros removed. For example:-

```
345                23.6789                180005
```

Numbers outside this range will be printed in exponential format to avoid losing significant figures. For example:-

```
4.87600E+8         1.58300E-46         1.60000E-19
```

In exponential format, up to six significant digits are displayed as a decimal number between 1.00000 and 9.99999, followed by an exponent section which indicates by what power of ten the decimal number should be multiplied to obtain the intended number.

Hence

4.87600E+8 would be	4.876×10^{8}	in scientific notation or
	487 600 000	in decimal notation
1.60000E-19 would be	1.6×10^{-19}	in scientific notation or
	0.000000000000000000016	in decimal

To be able to print the very large range of numbers calculated by BASIC, from 1.00000E-64 to 9.99999E+62, it should be clear that the use of the exponential form for printing them is imperative. Printing very large or very small numbers in decimal format would take up too much space and would be far less easy to understand than exponential notation. If the normal output format does not suit a particular purpose, then the number can be converted to a string and its format adjusted as desired using string functions. This technique is discussed in the section on formatting.

NUMBER FIELD

The field in which a number is printed is 12 characters wide. The positions are used for the following information.

POSITION	USE
1	Mantissa sign
2	Integer
3	Decimal point
4 - 8	Fractional part
9	E (indicating exponential form)
10	Exponent sign
11 - 12	Exponent

FORMATTING OUTPUT

Although the use of a comma to separate the items in a PRINT statement will result in the items being printed commencing in a new print zone each time, this simple formatting has two major shortcomings.

1. It may be desirable to use columns spaced more or less than thirteen printing spaces apart, and
2. When numbers are printed they are left justified into the print zones and for integers the digits of equal significance may not line up.

For example

```
342                              342
1        rather than               1
10000                          10000
15                                15
```

TAB

The first problem in setting up print columns different to the print zones provided can be solved using the TAB function. TAB is much like the tabulator key on a typewriter and causes the print position to be advanced to a specified column. Thus

```
10 PRINT TAB(20);"*"
```

will cause the print position to be advanced to the twentieth position from the left hand margin and an asterisk to be printed. If the print position is already past the TAB position specified, possibly as a result of a previous print statement, then no further tabbing will take place.

The TAB operand in the brackets can also be a numeric variable or a numeric expression.

```
10 FOR I=1 TO 10
20 PRINT I;TAB(I*10);
30 NEXT I
```

will cause

```
1         2         3         4         5         6
```

the numbers 1 to 6 to be printed, each ten print positions apart. (Refer to the section on FOR - NEXT loops if you are not familiar with these)

The second problem can be solved by translating the numeric variable into a string variable using the STR$ function. Once the number is in the form of a string of characters, it can be padded out with spaces to justify it appropriately.

```
10 REM ROUTINE TO RIGHT JUSTIFY A NUMBER BY ADDING SPACES
15 REM INTEGRAL NUMBERS ONLY
20 REM CONVERT NUMBER IN A TO STRING A$
30 LET A$=STR$(A)
40 REM ADD SPACES TILL LENGTH OF STRING >=8
50 IF LEN(A$)<8 THEN A$=" "+A$ : GOTO 50
60 PRINT A$
```

This routine will convert the numeric variable A into the string of numbers which would be printed by a PRINT statement in line 30, and then in line 50 adds leading spaces by string concatenation (see page 39) until the length of the string A$ is 8 characters. This will cause the previous list of numbers to be printed thus.

```
  342
    1
10000
   15
```

MATHEMATICAL OPERATORS

Four mathematical operations are performed by this BASIC. They are:-

+	Addition
-	Subtraction
*	Multiplication
/	Division

Addition and multiplication are straight forward floating point addition and multiplication of the two numbers involved. Subtraction will subtract the second expression from the first in the same way as it reads. Thus

```
10 LET A=5-4
```

will result in the variable A having a value of 1. Division is performed similarly. The first expression is divided by the second. Thus

```
20 LET A=200/2
```

will result in A having a value of 10.

A mathematical expression may contain more than one operator. Even a complicated expression like

```
30 LET A=B-4*5/6+C
```

will be resolved by BASIC provided B and C have previously been assigned a value. If B and C have not explicitly had a value assigned to them by an assignment statement, then they will be zero, the value to which all numeric variables are initialised at the start of a program. Numeric array variables however are not initialised to zero. This must be specifically done by the user if it is required.

PRIORITY OF OPERATIONS

A problem arises as the complexity of an expression increases. The greater the number of operators in an expression, the greater is the possibility of ambiguity as to the order in which the operations should be performed.

Line 30 for example, can be solved in a number of different ways which will lead to different solutions. The operations could be performed in order from right to left, from left to right, by addition and subtraction first followed by

multiplication and division, or by performing multiplication and division first then followed by addition and subtraction. Clearly the PRIORITY for performing each of these operations needs to be carefully defined. Additionally a means of specifying the order of operation when different from the standard, needs to be established.

BASIC has an inbuilt priority which defines that MULTIPLICATION and DIVISION are performed before any ADDITION or SUBTRACTION. The order in which multiplication and division are performed is from left to right. Thus

```
    3*4/2*5
=   12/2*5
=   6*5
=   30
```

The order in which addition and subtraction are performed is from left to right. Only the order of multiplication and division versus addition and subtraction will affect the answer. Using its priority rules, BASIC will interpret the statement

```
30 LET A=B-4*5/6+C
```

as

4 multiplied by 5 divided by 6, subtracted from B, plus C

PARENTHESES

PARENTHESES (brackets) can be used to group parts of an expression if it is desired to force BASIC to evaluate the expression in a different order. BASIC will evaluate the expressions in the innermost parentheses first and successively work outwards. Line 30 could be written as:-

```
30 LET A=(B((4*5)/6))+C
```

Here all the parentheses are redundant because they merely force BASIC to evaluate the expression as it would normally. However, if line 30 were bracketed like this:-

```
30 LET A=(B-4)*5/(6+C)
```

the answer would be completely different. BASIC is forced by the parentheses to evaluate (B-4) and (6+C) first and then to multiply (B-4) by 5 and then divide that answer by (6+C). The general rule is:-

IF IN DOUBT, USE PARENTHESES

They may not be needed, and although they will slow down the

execution of a program slightly, using parentheses will make the meaning of the expression much clearer.

For example:-

```
10 LET A=5*4-3*2          A=14
20 LET A=(5*4)-(3*2)      A=14
30 LET A=5*(4-3)*2        A=10
40 LET A=((5*4)-3)*2      A=34
50 LET A=(5*4-3)*2        A=34
```

Line 40 and 50 produce the same answer and are equivalent. The innermost parentheses are not required in line 40. Lines 10 and 20 also give the same answer. Here neither of the bracket pairs are required since they merely force the expression to be evaluated in the same way as it would normally.

BASIC actually scans the expression from left to right and pushes unresolved values onto an internal 'stack' until it can resolve them. The effect is as described previously, but the physical mechanism for achieving this is somewhat different.

ERRORS

If an expression evaluates to a value which is greater than 9.99999E 62 then an OVERFLOW ERROR will result.

STRING OPERATOR

Concatenation means joining together two or more string variables to form one value. The operator for the concatenation of two strings is the plus sign, '+'.

When concatenating strings, the first string expression may be any valid string, a string variable, a string constant, or a string expression. But the second and subsequent string expressions cannot include string functions such as LEFT$, MID$, RIGHT$ and STR$. Only a string constant or a string variable or CHR$ is permitted in this position. This restriction occurs because BASIC uses a buffer area to store the first string to be concatenated and performing a string function such as LEFT$ will also use this string buffer and destroy the first string to be concatenated. Line 10 is not permitted because it contains the string expression LEFT$ as the second section of the expression. A method of avoiding this construction is illustrated in line 20 below.

```
10 LET A$=B$+LEFT$(C$,3)              INVALID

20 LET Z$=LEFT$(C$,3) : LET A$=B$+Z$
```

Another example where three string expressions are concatenated to form one string is shown below.

```
10 LET A$="MARTIN"
20 LET B$="HOOD"
30 LET C$=A$+" "+B$
```

Here C$ becomes "MARTIN HOOD", the combination of the string variable A$ containing "MARTIN", the string constant " ", (a space) and the string variable B$ containing "HOOD". Strings may be concatenated up to a maximum length of 158 characters. However once a string is longer than 79 characters, it cannot be compared to another string or printed in the IMMEDIATE MODE. This is because it is longer than the string function buffer which is 79 characters long. It can still be printed by a print statement as part of a program. If any danger exists of a string being concatenated to a length greater than 79 characters, the length of the string can be tested using the LEN function, and precautions written into the program to prevent this from happening. 79 characters is a full line of text, so this should rarely be a limitation.

BRANCHING

One of the most powerful features available when using a computer, is the ability to repeat operations over and over again. A program 'loop' can be formed in BASIC using the GOTO instruction which transfers the execution of the program from the current line to the line specified in the GOTO statement. The line

```
100 GOTO 10
```

will cause BASIC to continue executing the program at line number 10. The following program will INPUT a number from the keyboard, calculate its square and square root and print out these values. At line 30 the GOTO instruction will cause the program to loop back to line 10 and go through the whole process again.

```
10 INPUT"NUMBER";A
20 PRINT A*A,SQR(A)
30 GOTO 10
```

In this program the loop will continue forever. It will never terminate. You can stop it by pressing the ESCape key while it is expecting an input, or by pressing the BREAK key while it is printing and abort the program. However a preterable way to exit from an endless loop like this is to include a conditional branch out of the loop. This is discussed in the next section.

GOTO

Although BASIC normally ignores all spaces, one place where it does not do so is in KEYWORDS. Therefore GOTO must be typed with no space between the GO and the TO. GO TO is not permitted and will cause a SYNTAX ERROR.

If the line number specified in a GOTO statement does not exist then a NOGO ERROR will result.

IMPLIED GOTO

The GOTO instruction is not required if the line number follows an IF-THEN statement.

IF A=B THEN 1000 is the same as IF A=B THEN GOTO 1000

CONDITIONAL STATEMENTS AND RELATIONAL OPERATORS

BASIC allows statements to be executed dependent on whether a specified condition is tested to be true. The form is

```
IF <condition> THEN <line number>
IF <condition>[THEN]<statement>
```

In the second form the 'THEN' is optional. The following statements are equivalent.

```
IF A=B PRINT A    is the same as    IF A=B THEN PRINT A
```

The <condition> section tests the truth of a relation between two expressions. The expression can be a CONSTANT, a VARIABLE, or the solution of an expression involving both CONSTANTS and VARIABLES.

The relations which can be tested are

```
=    the expressions are equal
<    expression 1 is LESS THAN  expression 2
>    expression 1 is GREATER THAN expression 2
<=   expression 1 is LESS THAN OR EQUAL TO expression 2
>=   expression 1 is GREATER THAN OR EQUAL TO  expression 2
<>   expression 1 is NOT EQUAL TO  expression 2
```

Note: =, <, and > may be in any order so >= is the same as =>.

The expressions compared in a conditional statement may be either STRING expressions or NUMERIC expressions. However it does not make sense to compare expressions of unlike type so a STRING expression can only be compared with another STRING expression and a NUMERIC expression can only be compared with another NUMERIC expressions. Only expression of like type can be compared.

Each of the following statements are valid

```
10 IF A$ = B$ THEN 200
20 IF B = 1.4 THEN 210
30 IF SIN(2+B) = COS(A-4) THEN PRINT "REALLY?"
```

but

```
40 IF B = A$ THEN GOTO 230          INVALID
```

is not valid because variables of unlike type are being compared, and will cause a TYPE ERROR.

The statement following the "THEN" can be any valid BASIC statement. All of the following are valid examples.

```
10 IF A < B THEN STOP
20 IF A$ = "YES" THEN A=A+1
30 IF B$ = LEFT$(C$,1) THEN GOSUB 2000
40 IF A >= 100 THEN PRINT "NUMBER TOO LARGE"
```

WHAT HAPPENS WHEN THE TEST FAILS

If the relation tested is NOT TRUE, the program continues at the next numbered BASIC line. Neither the THEN expression NOR any other statement on the rest of a multiple statement line will be executed. Some other versions of BASIC will execute the other statements in a multiple statement line if the test fails, however this is less useful than the method used here because it demands that more GOTO statements be used to segment the flow of the program. For example

```
10 IF A=B THEN PRINT"A=B":LET A=1:INPUT B
20 ..... program continued
```

will test for the equality of A and B and if it is true will print the message saying so, set A to 1 and input a new value for B. A BASIC which continued to execute the subsequent statement on a multiple statement line even after the test had failed, would require the following.

```
10 IF A<>B THEN GOTO 20
12 PRINT"A=B": LET A=1: INPUT B
20 ..... program continued
```

which is a little more complicated and is less easy to follow than the previous example.

This feature is also useful for converting IF statements involving AND and OR

```
IF A=1 OR B=2 OR C=3 THEN PRINT X
```

can be achieved using the following lines.

```
100 IF A=1 THEN PRINT X : GOTO 130
110 IF B=2 THEN PRINT X : GOTO 130
120 IF C=3 THEN PRINT X : GOTO 130
130 ........
```

Similarly

```
IF A=1 AND B=2 AND C=3 THEN PRINT X
```

could be replaced by

```
IF A=1 IF B=2 IF C=3 PRINT X
```

STRING COMPARISONS

STRINGS may also be compared using these relational operators. The ASCII value of each of the characters in each STRING is compared until one character is found to be greater in ASCII value than the corresponding character in the other string. If the strings are not equal in length and if all else is equal, then the longer of the two strings is deemed to be greater than the shorter string. For two STRINGS to be EQUAL they must be equal in length AND contain identical characters.

Using STRING comparisons to sort a list of words into alphabetic order can lead to apparent anomalies if punctuation or numbers are included. The sorting order will be the same as the table of ASCII equivalents. (see appendix c) A space is the character with the smallest ASCII value and is considered to be the 'smallest' character in a comparison. Punctuation is followed by numbers in priority, then upper case letters and finally lower case letters in ascending order of magnitude.

Some examples of string comparisons are illustrated below

Example 1.

```
5  REM  PROGRAM TO ILLUSTRATE STRING COMPARISONS
10 INPUT "DO YOU WISH TO CONTINUE";Z$
20 IF LEFT(Z$,1)="N" THEN STOP
```

Example 2

```
50 INPUT "NUMBER";N$
60 IF ASC(N$)<48 THEN GOTO 100
80 N=VAL(N$)
90 GOTO 120
100 PRINT "NUMBERS ONLY PLEASE"
110 GOTO 50
120 ......program continued
```

Note:- String variables containing more than 79 characters cannot be compared.

FOR - TO - NEXT LOOPS

As was mentioned in the section of this manual which deals with the branch instruction GOTO, one of the most useful features of a computer is its ability to perform a repetitive task many times. A program which repeats a set of instructions is said to 'loop'. A 'loop' can be made using an IF - THEN - GOTO statement. For example the program

```
10 LET A=0
20 PRINT "-";
30 LET A=A+1
40 IF A<= 80 THEN GOTO 20
9999 END
```

will print a hyphen '-', increment the variable A, and then if A is less than or equal to 80, will loop to do the procedure again. This will continue until A is greater than 80. In effect, the program will print a line containing 80 hyphens.

Because this type of loop stucture where the number of times the loop is to be executed is known in advance, is so commonly used in programs, BASIC has an instruction specifically for making a loop like this within a program. This is called a FOR - NEXT loop. The program in the previous example could be rewritten as follows using a FOR - NEXT loop.

```
10 FOR A=1 TO 80
20 PRINT "-";
30 NEXT A
9999 END
```

This produces exactly the same result as the first example, but is much simpler. The FOR statement assigns a starting value to the numeric variable specified, in this case A, and stores the terminating value,as well as a pointer to the position of the FOR in the program. Initially A is made equal to 1 and the program continues until a NEXT is encountered. The NEXT must specify the same variable as the FOR statement or a NEXT ERROR will result. The NEXT statement increments the variable, and tests to see if it is greater than the terminating value. If the value of the variable is NOT greater, then the program will loop back to the statement following the original FOR statement and execute all the statements between it and the NEXT yet again. If the value of the variable IS greater than the terminating value, then the program will continue execution at the next statement after the NEXT, either at the following line, or at the statement after the NEXT on a multiple statement line.

The FOR-NEXT loop is the fastest way of making a loop structure in this BASIC. Since the test is performed at the NEXT, the loop will always be executed once, even if the test is

initially false.

STEP

A FOR-NEXT loop does not necessarily have to increment by 1 each time around the loop. It can increment or decrement by any number. This does not need to be an INTEGER. The part of the instruction which defines the size of the increment or decrement is STEP. For example

```
10 FOR I=80 TO 40 STEP -1.5
..........
..........
50 NEXT I
```

will execute the loop beginning with I = 80 and then DECREMENTING then value of I by 1.5 each time until I is LESS THAN 40. Nor do the beginning and terminating values need to be integers or constants. The limits can be any resolvable arithmetic expression. But since the terminating value and the STEP size are stored when the loop is first entered, these cannot be altered within the loop. The value of the variable MAY be altered within the loop. In fact it can be treated just like it was a normal program variable. This can be particularly useful if it is desired to exit a FOR - NEXT loop prematurely.

EXITING FOR - NEXT LOOPS

It is not advisable to prematurely exit from a FOR - NEXT loop by using a GOTO statement. This is because BASIC pushes details about the FOR - NEXT loop onto an internal stack and if the loop is exited by a GOTO statement, sixteen bytes of information are left pushed up onto the stack. If this practice is repeated a number of times it will quickly fill up the available stack space, and a STACK OVERFLOW ERROR will result. The following is NOT RECOMMENDED because it will fill up the stack.

```
10 FOR I=1 TO 10
20 IF A<B THEN GOTO 50
30 ...some operations
40 NEXT I
50 ...more program...
```

When B is tested to be greater than A the FOR - NEXT loop will be exited by the GOTO instruction, but the details are left on the stack.

If it is desired to exit from a FOR - NEXT loop, this can be done by changing the value of the variable within the loop so

that the terminating condition is satisfied. A LET instruction could be used to set the variable to a very large value in the case of a positive STEP or a large negative value in the case of a negative STEP.

NESTING FOR - NEXT LOOPS

FOR - NEXT loops may be NESTED, that is one complete loop may be contained within another. For example

```
10 FOR I=1 TO N
20 FOR J=10 TO 1
30 ...........
40 ...........
50 NEXT J
60 NEXT I
```

When nested, each FOR - NEXT loop MUST use a different variable as the counter, and the inner loop must be completely within the outer loop. The following is not permitted because the loop using J as the variable is not completely within the loop using I as a variable. The loops are 'misnested' and a NEXT ERROR will result.

```
10 FOR K = 100 TO -10 STEP -5          INVALID
20 FOR L = 1 TO 10
30 ........
40 ........
50 NEXT K
60 NEXT L
```

The maximum depth to which FOR-NEXT loops can be nested is 7, but this will only be the case if no arithmetic expressions are to be performed. In practice the maximum depth of nest will be somewhat less. However it is a very rare program that requires more than three or four FOR - NEXT loops to be nested, so it is unlikely to be a problem.

SUBROUTINES

Some more complicated programs have parts which are more appropriately treated as a separate section and then fitted back into the main program. This is especially the case if the section of program will be used a number of times at different points in the main program.

This smaller program within a larger one is called a SUBROUTINE, or SUBPROGRAM.

To call a subroutine from the main program, the instruction GOSUB <line number> is used. The program continues at the specified line number until a RETURN instruction is encountered. If the specified line number does not exist, a NOGO ERROR will result.

The RETURN instruction causes the program to resume at the next statement after the original subroutine call. If the GOSUB is part of a multiple statement line, then the statement executed following a RETURN is the statement after the GOSUB on the multiple statement line. For example

```
10 GOSUB 6000 : PRINT "VOLUME = ";V;" CUBIC UNITS"
20 STOP
.........
.........
6000 REM SUBROUTINE TO CALCULATE THE VOLUME OF A CYLINDER
6010 REM OF RADIUS R AND LENGTH L
6020 REM
6030 LET V=3.14159*R*R*L
6040 RETURN
```

will transfer execution of the program to line 6000, calculate the volume of the cylinder at line 6030 and at line 6040 will return to the PRINT statement following the GOSUB on line 10.

NESTING SUBROUTINES

Subroutines may be nested, that is one subroutine may call another subroutine and that subroutine may call yet another subroutine and so on. The number of times this can occur before a RETURN is executed depends on the usage of the internal stack by FOR - NEXT loops and the complexity of numeric expressions to be resolved, but should normally be far more than will ever be needed. If the capacity of the stack is exceeded by any of these reasons, including too many GOSUBS without a RETURN, a STACK OVERFLOW ERROR will result. The occurrence of a RETURN ERROR means that a RETURN has been encountered without a subroutine having been called.

POKE

POKE(A,B) will store the BINARY equivalent of the decimal number B into the memory location indicated by the decimal number A. The value of the number to be stored in memory must be between Ø and 255, (Ø and FF hex) and the address in decimal must lie between Ø and 32767 (Ø and 7FFF hex). As with all other BASIC statements, A and B can be any valid numeric expression which yields a result within the allowed range.

NOTE: POKEing around in memory can be very dangerous. You could accidently change a memory location in the BASIC interpreter (if it is located in read write memory) which would subsequently cause unpredictable results, or if you are lucky, the system to crash. Be very careful and make absolutely sure that you have the correct memory location (in decimal) before POKEing at it.

CALL

The function CALL(A) will branch to a machine language subroutine commencing at memory location A. As previously, the value of A is in decimal, not hexadecimal or octal. Make absolutely sure that you have calculated the starting address of the subroutine correctly, and that the routine does actually exist and that it terminates with a RETURN, otherwise you will BOMB out of BASIC. Be careful!!

FUNCTIONS

Functions supported by this BASIC are listed below.

ABS, INT, SQR, COS, SIN, LOG, and EXP

are NUMERIC functions and act on a NUMERIC expression to produce a NUMERIC result.

LEFT$,MID$ and RIGHT$

are STRING functions. They act on a STRING expression and produce a string as a result. They each return a specified section of the original string.

LEN,VAL and ASC

are functions which act on a STRING expression but produce a NUMERIC result.

STR$ and CHR$

produce a STRING as a result of a NUMERIC expression.

PEEK

is a special function used to examine the contents of a specific location in memory.

RND(1)

returns a random number between Ø and 1.

ACCURACY OF FUNCTIONS

The functions SQR, COS, SIN, LOG, and EXP will produce a result which is accurate to five signifcant figures. Although BASIC actually produces a result which contains six figures, the last digit is uncertain and should be rounded, not truncated, to give the correct value.

ABS

```
10 LET B=ABS(A)
```

makes B equal to the ABSOLUTE VALUE of A. If A is -1.4 then B will be 1.4. If A is +20 then B will also be 20. The ABS function deletes any negative sign from a numeric variable.

INT

```
10 LET B=INT(A)
```

will return the INTEGER portion of the variable A. Thus if A is 1.548 then B will be 1. If A is 106.998 then B will be 106. Note particularly that if A is negative, -8.54 for example, then B will be -8. Some other versions of BASIC would make this -9.

SQR

```
10 LET B=SQR(A)
```

will make B equal to the SQUARE ROOT of A. Thus if A is 2 then B will be 1.4142 If A is negative, an ARG ERROR will result.

COS

```
10 LET B=COS(A)
```

returns the COSINE of the angle A, when A is expressed in RADIANS. An angle in degrees can be converted to radians by multiplying by the factor pi/180 or 0.0174532. Thus

```
10 LET B=COS(A*3.14159 /180)
```

will produce the correct answer if A is the angle in degrees rather than the angle in radians. The angle in radians must have a value between 0 and pi. ie. 0 degrees and 180 degrees.

SIN

```
10 LET B=SIN(A)
```

will return the SINE of the angle A where A is again expressed in RADIANS rather than degrees. For the SIN function, the value of the expression it is evaluating must lie between -pi/2 and +pi/2 ie. -90 degrees and +90 degrees.

LOG

```
10 LET B=LOG(A)
```

will make B equal to the logarithm to base 10 of A. This is different to most other versions of BASIC which will return the logarithm to base e. LOG to base 10 is considered to be more useful than LOG to base e, since most calculations involving logarithms will be in base 10. If logarithms to base e are required, multiplying the LOG to base 10 by 2.30258, will provide the equivalent of LOG e (A). If A is zero or negative, an ARG ERROR will result.

```
LOG e (A)  =  2.30258 * LOG 10 (A)
```

EXP

```
10 LET B=EXP(A)
```

will make B equal to ten raised to the power A. This is also different to most other versions of BASIC which return e raised to the power A. Again exponents to base 10 are often more convenient to use in calculations and EXP(A) is consistent with LOG(A) since they are both calculated to base 10. EXP(A) is the same as ANTILOG.

If it is required to calculate EXP e (A), use the expression

```
B=EXP(A/2.30258)
```

A combination of LOG and EXP can be used to calculate the value of a number raised to a power. For example, 5 raised to the power of 2.5 can be calculated by the following expression.

```
10 LET A=EXP( LOG(5) * 2.5)
```

yielding an answer of 55.902. This combination of functions yields a result which is only accurate to four significant figures.

LEFT$ - MID$ - RIGHT$

These three string functions return a section of the specified string. As the names suggest, LEFT$ returns the LEFT hand portion of the string, MID$ returns the middle portion and RIGHT$ returns the right hand portion.

LEFT$

```
10 LET B$=LEFT$(A$,N)
```

will make B$ equal to the N leftmost characters in the string A$. If A$ ="FREDERICK" then

LEFT$(A$,4) would be the string "FRED"

RIGHT$

```
10 LET B$=RIGHT$(A$,N)
```

will make B$ equal to the N rightmost characters of the string A$. If as before A$ = "FREDERICK" then

RIGHT$(A$,4) would be the string "RICK"

MID$

```
10 LET B$=MID$(A$,N,M)
```

will make B$ equal to the M middle characters starting from the Nth character. Thus

MID$(A$,4,3) would be the string "DER"

LEN

LEN returns the LENGTH of a string. Thus

```
10 LET A=LEN("FREDERICK")
```

would make the numeric variable A equal to 9, the length of the string "FREDERICK". LEN can also operate on a string variable.

```
10  LET A$="ONE TWO THREE"
20  LET A=LEN(A$)
```

A would be 13, the length of the string in A$, including any imbedded spaces.

Suppose you wanted to print the section of a string from the fourth character to the end. The following expression will print all but the first four characters, irrespective of the length of A$.

```
10 LET L=LEN(A$):PRINT RIGHT$(A$,L-4)
```

Note: The statement RIGHT$(A$,LEN(A$)-4) WILL NOT WORK! This is because nesting of string expressions is not permitted since the LEN function uses the same work buffer as the RIGHT$ function. Executing the LEN function will scramble the RIGHT$ contents in the buffer.

VAL

VAL returns a numeric variable equal to the number contained in a string expression. The lines

```
10 INPUT"NUMBER";A$
20 LET A=VAL(A$)
```

would input a number AS A STRING, "1234.5" for example, at line 10 and then the VAL function translates the contents of the string variable A$ into the numeric variable A. Whereas you cannot perform any arithmetic operations on the string variable A$, you can do so to the translated numeric variable A.

ASC

ASC returns the decimal equivalent (NOT octal or hexadecimal!) of the ASCII value of the FIRST character in the specified string. See appendix C for a table of ASCII - DECIMAL values. Thus if the string variable A$ was "FRED" then ASC(A$) would be 70, because 70 is the decimal value of "F", the first

character in the string.

The ASC function can be used to check for numeric or alphabetic input. For example

```
10 INPUT"NUMBER";A$
20 IF ASC(A$)>57 THEN 50
30 IF ASC(A$)<48 THEN 50
40 PRINT A$;" IS A NUMBER": GOTO 100
50 PRINT A$;" IS NOT A NUMBER"
100 END
```

If the ASCII value is less than 48 or greater than 57 then the character is not numeric. Note also that the use of the instruction GOTO is optional after a THEN and is omitted in lines 20 and 30.

The following program uses the ASC function and its inverse CHR$ to convert a lower case letter into upper case

```
20 IF ASC(A$)>=96 THEN Z=ASC(A$)-32 : A$=CHR$(Z)
```

If the first character in A$ is 96 or greater, ie. lower case then 32 is subtracted from the ASCII value, and that value translated back into a character by the CHR$ function.

STR$

STR$ is the inverse function of VAL. It translates a numeric expression into the string which would be printed by a PRINT instruction.

```
10 LET A=10
20 LET A$=STR$(A)
```

would convert the numeric variable A, which has a value of 10, into the string variable A$ containing the string " 10".

As a string, the individual characters can be manipulated to change the format of a number. This cannot be done to a numeric variable. An example of this is contained in the section on I/O statements where the STR$ function is used to justify a number.

CHR$

CHR$ is the inverse function of ASC. CHR$ converts an INTEGER into its ASCII equivalent.

```
CHR$(49) is the string "1"
CHR$(65) is the string "A"
```

CHR$ returns a string only one character in length. It can be used to output non-printing characters such as nulls, form feeds, bell etc. For example, when using a printer, it is often necessary to output a series of null characters at the end of each line to give the print head time to return to the left hand side of the page. The statements

```
10 FOR I=1 TO 10 : PRINT CHR$(0); : NEXT I
```

will output ten nulls (ASCII 0) which the printer will ignore, but will fill in time until it is ready to continue printing.

The characters between 128 and 255 can be used to display inverse characters if your system is equipped with a DG640 memory mapped display. A table of decimal equivalents of ASCII characters is provided in appendix c..

RND

```
10 LET B=RND(X)
```

will return a PSEUDO-RANDOM NUMBER between 0 and 1. The value of X does not matter as it is merely a dummy argument for the RND function. If it is desired to obtain a reproducable sequence of pseudo-random numbers, this can be achieved by setting the seed value to a particular value before using the RND(X) function the first time. The two memory locations containing the seed for the random number are 1901 and 1902 (decimal). POKEing a specific value, say 0, into both of these locations will cause the same sequence of random numbers to be produced each time. For example

```
100 POKE(1901,0) : POKE(1902,0)
```

will initialise the random number seed to 0 and cause a specific series of pseudo-random numbers to be produced.

PEEK

PEEK returns, in decimal, not octal or hexadecimal, the contents of the address specified. Thus

```
10 PRINT PEEK(29)
```

will print '4', which is the decimal equivalent of memory location 001D (hex), if you are using PIPBUG or BINBUG.

Appendix a.

DEFINITIONS

EXPR is any expression either string or numeric.

VAR is any variable, numeric, string, or array.

NEXPR is any expression which may be replaced by a number. It may be made up of variables, constants, operators and parentheses e.g. A*2/(5E4/B - 27.2)

AVAR is any array variable A(NEXPR) thru Z(NEXPR)

NVAR is any simple variable indicated by a letter A thru Z or a single letter followed by a digit Ø thru 7

CONST is a numeric constant consisting of at least 1 decimal number with an optional decimal point, optionally followed by an exponent. e.g. 23, 1.23, Ø.3, 1.45E2, 1.3E-4, 1E5

OPER is one of the operators + - * / representing respectively addition, subtraction, multiplication and division.

$EXP is any expression which may be replaced by a string. It may be a string variable, string constant, string function, or a string concatenation.

$VAR is any string variable A$ thru Z$.

$CONST is a string constant which consists of up to 79 characters enclosed between quotes e.g. "THIS IS A STRING CONSTANT"

$ARRAY is a string array A$(NEXPR) thru Z$(NEXPR) the maximum number of elements is 255 (from Ø to 254)

$OPER is a string concatenation indicated by $EXP+$EXP+... while the first $EXP may be any valid expression, the second and subsequent may only be $VAR or $CONST or CHR$. i.e. the string functions LEFT$,RIGHT$,MID$,STR$ are not permitted. Concatenated strings may be used as arguments in functions.

RELN is one of the following relational operators =, <, >, <=, =<, >=, =>, <>, ><.

line No. is a line number between 1 and 9999.

STMT is any valid BASIC statement.

SUMMARY OF COMMANDS

COMMAND	EXAMPLES	NOTES	REFERENCE
C[ONTINUE]	C CONT	Continue executing the program. Program must not have been altered between BREAK or STOP and CONTINUE.	17
L[IST][#n]... ...[,line No]... ...[,line No]	L LIST L20 L#1,20 LIST#N,20,40	List entire program. List entire program. List line 20 only. List Line 20 to Port#1. List lines 20 to 40 inclusive to Port#N.	11,14
LO[AD]	LOAD LO	Load a previously SAVEd program.	19
N[EW]	N	Delete any program currently in memory.	11
O[LD]	O OLD	Reset all text pointers and zero non-array variables.	11
R[UN]	R RUN	Run the program currently in memory.	12
SA[VE]	SAVE SA	Save the program	18
S[IZE]	S SIZE	Display the size of the program and variable storage.	18

SUMMARY OF KEYWORDS

Sections of STATEMENTS enclosed in square brackets are optional.

KEYWORD	EXAMPLES	NOTES	REFERENCE
CALL	[line No.] CALL(<NEXPR>)	Ø <= <NEXPR> <= 32767	48
	10 CALL(1096) CALL(Ø) 20 CALL(A+B)		
DATA	line No. DATA <CONST> , <CONST> , ... <$CONST> <$CONST>	No multiple statement may follow on same line. See also READ.	25
	10 DATA FRED,45,10E-4		
END	[line No.] END		15
	9999 END		
FOR	[line No.] FOR <NVAR> = <NEXPR1> TO <NEXPR2> [STEP <NEXPR3>]	See also NEXT.	44
	10 FOR I1 = 10 TO 1 STEP -1 20 FOR Z7 = A TO B 30 FOR D(1) = 9 TO Z STEP X(2)-4		
GOTO	[line No.] GOTO <line No.>		40
	10 GOTO 825 GOTO 100		
GOSUB	[line No.] GOSUB <line No.>	See also RETURN	47
	90 GOSUB 8000 GOSUB 120		

IF-THEN 41

```
                  <NEXPR>          <NEXPR>
[line No.] IF ------- <RELN> -------  THEN  <line No.>
                  <$EXPR>          <$EXPR> [THEN] <STMT>
```

```
100 IF A<B THEN 1000
200 IF A$="Y" PRINT "WHAT?"
300 IF A>=B IF A<C PRINT"A IS BETWEEN B AND C"
```

INPUT 27-30

```
line No. INPUT [#<NEXPR>,] <$CONST>,
                           ...<VAR>,<VAR>...
```

```
10 INPUT A$
20 INPUT#3,B,C,D
30 INPUT#A+1,B,C$
```

LET 24

```
                  <NVAR>
                  <AVAR>          <NEXPR>
[line No.] [LET] -------  =  -------
                  <$VAR>          <$EXPR>
                  <$ARRAY>
```

```
LET A=B
100 A(N) = 64
150 Q$ ="FRED"
200 R$ = LEFT$(A$,6)
```

NEXT 44

```
[line No.] NEXT <NVAR>
```

See also FOR

```
100 NEXT I
150 NEXT Z(2)
```

POKE 48

```
[line No.] POKE( <NEXPR1> , <NEXPR2> )
                  0 <= <NEXPR1> <= 32767
                  0 <= <NEXPR2> <= 255
```

```
10 POKE (1901,0)
POKE (1902,198)
```

PRINT 31-35

```
                                          <;>
[line No.] PR[INT] [#<NEXPR>,] <EXPR>     <EXPR> ..
                                          <,>
```

```
PRINT SQR(2)
100 PRINT A
200 PRINT#2,A,B$
300 PRINT#A/3,"HELLO";N$
```

READ line no. READ <VAR> , <VAR> , ... 25
See also DATA

100 READ A$,B,C

REM [line No.] REM <comment> 15

200 REM THIS IS A COMMENT LINE. IT IS IGNORED.

RETURN [line No.] RETURN 47
See also GOSUB

10 RETURN

STOP line No. STOP 17

6000 STOP

SUMMARY OF FUNCTIONS

FUNCTION	EXAMPLES	NOTES	REFERENCE
ABS	ABS(<NEXPR>)		50
	10 LET A=ABS(Z+Y-3) 20 PRINT ABS(C) 30 A(X) = SQR(ABS(3-A))		
ASC	ASC(<$EXPR>)	First character only.	53
	10 A$ = ASC(RIGHT$(A$,3)) 20 PRINT ASC(Q$)		
COS	COS(<NEXPR>)	0 <= <NEXPR> <= pi	50
	10 M=COS(A+C*3) 20 PRINT COS(F)		
CHR$	CHR$(<NEXPR>)	0 <= <NEXPR> <= 255	54
	10 LET A$ = CHR$(N+M) 20 PRINT#1,CHR$(0)		
EXP	EXP(<NEXPR>)		51
	10 A = EXP(A-1) 20 PRINT EXP(ABS(A)+1)		
INT	INT(<NEXPR>)	Note:- INT(-8.5) =-8	50
	10 A = INT(A-3*4) 20 PRINT INT(RND(1)*100+1)		
LOG	LOG(<NEXPR>)	<NEXPR> > 0	51
	10 A = LOG(3) 20 PRINT LOG(A1) 30 A(I)=LOG(VAL(MID$(A$,3,2)))		
LEFT$	LEFT$(<$EXPR> , <NEXPR>)		52
	10 PRINT LEFT$(A$,6) 20 A=LEFT$(B$,N-1) 30 IF LEFT$(Z$,1)="Y" THEN 1000		

LEN LEN(<$EXPR>) 53

```
A = LEN(A$)
10 PRINT TAB(LEN(A$))
```

MID$ MID$(<$EXPR> , <NEXPR1> , <NEXPR2>) 52

```
10 A$ = MID$(A$,64,2)
20 PRINT MID$(H$,N,M)
```

PEEK PEEK(<NEXPR>) 0 <= <NEXPR> <= 32767 55

```
10 A = PEEK(1901)
20 PRINT PEEK(N+5)
```

RIGHT$ RIGHT$(<$EXPR> , <NEXPR>) 52

```
10 A$ = RIGHT$(A$,3)
20 PRINT RIGHT$(Z$,B+4)
```

RND RND(1) 55

```
10 PRINT RND(1)
20 LET A = INT(RND(1)*10+1)
```

SIN SIN(<NEXPR>) -pi/2 <= <NEXPR> <= pi/2 51

```
10 S = SIN(A+B*N)
20 PRINT SIN(Q)
```

SQR SQR(<NEXPR>) <NEXPR> >=0 50

```
10 LET A = SQR(B*4)
PRINT SQR(2)
20 A = SQR(INT(B-4))
```

STR$ STR$(<NEXPR>) 54

```
10 A$=STR$(A(50))
20 PRINT LEFT$(STR$(N),4)
```

TAB	TAB(<NEXPR>)	1 <= <NEXPR> <= 255	34

```
10 TAB(10)
20 PRINT TAB(I*5+4)
```

VAL	VAL(<$EXPR>)	<$EXPR> must be a valid number.	53

```
10 IF VAL(A$)>999 PRINT "WHAT?"
20 A = VAL(Z$)
```

GENERAL

BLANKS — All blanks are ignored, except in keywords, and may be included to improve readability or deleted to save space.

LINE NUMBERS — Each valid line of the BASIC program commences with a line number between 1 and 9999, and all commands are in upper case.

MULTIPLE STATEMENTS — Multiple statements may be on the same line if they are separated by a colon ":". A REMark statement must be the last on a line. No other statement may follow a DATA statement.

NUMERIC VARIABLES — This version of BASIC allows numeric variables identified by a letter A through Z or a letter followed by a digit 0 through 7. Numeric variables are accurate to 6 figures and may be in the range 1E-64 to 9.99999E62.

NUMERIC ARRAYS — Up to 26 arrays A through Z are allowed.

STRING VARIABLES — There are also 26 permissible string variables, identified by A$ through Z$. String variables may contain up to 158 characters, although strings of more than 79 characters may not be compared in IF statements or manipulated in IMMEDIATE mode.

STRING ARRAYS — Up to 26 string arrays A$ through Z$ are allowed, each of which may have up to 255 elements.

Appendix b

ERROR MESSAGES

If BASIC discovers an error while executing a program, it will display one of the following error messages and list the line containing the error. Execution of the program will cease and BASIC will return to the editor mode.

SYNTAX ERROR — The statement does not conform to the correct syntax.

STACK OVERFLOW ERROR — This indicates one, or a combination, of the following

a. expression too complicated.
b. FOR - NEXT loops too deeply nested.
c. GOSUBs too deeply nested.

OVERFLOW ERROR — Value of expression exceeds 9.99999E62 or a string contains more than 158 characters.

ARG ERROR — attempt to divide by zero, find the LOG of a negative number, find the LOG of zero, or find the square root of a negative number.

NEXT ERROR — A NEXT statement was encountered without a corresponding FOR or improperly nested FOR - NEXT loops.

NOGO ERROR — Line number not found by GOTO, GOSUB or IF-THEN statement.

RETURN ERROR — RETURN encountered without having performed a GOSUB.

DATA ERROR — Attempt to READ past the end of the last DATA statement.

MEMORY ERROR — Available Read/Write memory area has been exceeded.

DIM ERROR	Attempt made to redimension an array.
FILE ERROR	Invalid program file or attempt made to CONTinue after modifying a program.
BUFFER ERROR	More than 79 characters were entered on the one line.
TYPE ERROR	Mismatch between variable and expression type.

Appendix c.

ASCII-HEXADECIMAL-DECIMAL TABLE

ASCII	HEX	DECIMAL	ASCII	HEX	DECIMAL	ASCII	HEX	DECIMAL
NUL	00	0	+	2B	43	V	56	86
SOH	01	1	,	2C	44	W	57	87
STX	02	2	-	2D	45	X	58	88
ETX	03	3	.	2E	46	Y	59	89
EOT	04	4	/	2F	47	Z	5A	90
END	05	5	0	30	48	[	5B	91
ACK	06	6	1	31	49	\	5C	92
BEL	07	7	2	32	50	]	5D	93
BS	08	8	3	33	51	^	5E	94
HT	09	9	4	34	52	_	5F	95
LF	0A	10	4	35	53	`	60	96
VT	0B	11	6	36	54	a	61	97
FF	0C	12	7	37	55	b	62	98
CR	0D	13	8	38	56	c	63	99
SO	0E	14	9	39	57	d	64	100
SI	0F	15	:	3A	58	e	65	101
DLE	10	16	;	3B	59	f	66	102
DC1	11	17	<	3C	60	g	67	103
DC2	12	18	=	3D	61	h	68	104
DC3	13	19	>	3E	62	i	69	105
DC4	14	20	?	3F	63	j	6A	106
NAK	15	21	@	40	64	k	6B	107
SYN	16	22	A	41	65	l	6C	108
ETB	17	23	B	42	66	m	6D	109
CAN	18	24	C	43	67	n	6E	110
EM	19	25	D	44	68	o	6F	111
SUB	1A	26	E	45	69	p	70	112
ESC	1B	27	F	46	70	q	71	113
FS	1C	28	G	47	71	r	72	114
GS	1D	29	H	48	72	s	73	115
RS	1E	30	I	49	73	t	74	116
US	1F	31	J	4A	74	u	75	117
SP	20	32	K	4B	75	v	76	118
!	21	33	L	4C	76	w	77	119
"	22	34	M	4D	77	x	78	120
#	23	35	N	4E	78	y	79	121
$	24	36	O	4F	79	z	7A	122
%	25	37	P	50	80	{	7B	123
&	26	38	Q	51	81	|	7C	124
'	27	39	R	52	82	}	7D	125
(	28	40	S	53	83	~	7E	126
)	29	41	T	54	84	DEL	7F	127
*	2A	42	U	55	85			

Appendix d.

MEMORY MAP

ADDRESS	DETAILS
Variable	Top of memory
Variable	Storage for STRINGS, STRING ARRAYS, STRING ARRAY LENGTHS, NUMERIC ARRAYS.
2000 +	User source program storage
159B - 1FFF	BASIC INTERPRETER
1477 - 159A	Personality module
0800 - 1476	BASIC INTERPRETER
0800	HARD START ENTRY POINT
0500 - 07FF	BASIC SCRATCHPAD READ WRITE MEMORY
0440 - 04FF	UNUSED BY BASIC
0400 - 043F	PIPBUG SCRATCHPAD READ/WRITE MEMORY
0000 - 03FF	PIPBUG OR BINBUG MONITOR ROM

APPENDIX e.

IMPORTANT LOCATIONS

$500-$57F		Line input buffer
$749,A	PSOF	Pointer to start of BASIC source program
$74B,C	PEOF	Pointer to end of BASIC source program
$76D,E	RNDM	Seed for random number generator
$800	BEGN	Hard start address of BASIC
$80D	ERRB	SOFT START ENTRY
$1477,8	SOP	Start of program storage pointer

TABLE OF I/O ADDRESS CONSTANTS

$1479,A	Channel	0 INPUT routine
$147B,C	Channel	0 OUTPUT routine
$147D,E	Channel	1 INPUT routine
$147F,80	Channel	1 OUTPUT routine
$1481,2	Channel	2 INPUT routine
$1483,4	Channel	2 OUTPUT routine
$1485,6	Channel	3 INPUT routine
$1487,8	Channel	3 OUTPUT routine

APPENDIX f.

I/O CHANNELS

The versatility of this BASIC is greatly enhanced by the provision of four sets of Input/Output 'channels'. By including an optional channel number in one of a range of I/O statements, data may be input from, or output to, up to four separate devices. The device used can be selected under program control since the channel number may be specified by an expression or a numeric variable.

As supplied, channels 1,2 and 3 all vector to the I/O routines in the monitor, CHIN and COUT. Channel #Ø communicates with the keyboard and TVT display via a set of driver routines within the 'personality module'. To use channels 1, 2 or 3 the user must provide driver subroutines, one to input a character from the device, another to output a character to the device, and alter the address constants contained in the table beginning at $1479 to point to the start of these routines.

Fundamentally these routines must emulate the PIPBUG I/O routines. An INPUT subroutine should return the character input in register Ø with the parity bit Ø. An OUTPUT subroutine should output the ASCII character contained in register Ø to the device.

Both the routines must terminate with an unconditional return from subroutine, RETC,UN ($17), and must not alter registers 1, 2 or 3. The alternate register set R4, R5, and R6 may be used within the routine provided the normal register set is reselected before returning. The LCOM bit of the PSW must also be preserved set to 1 by the routine.

WHERE TO PUT IT

Space may be made for your I/O routines by moving the start of the BASIC source file storage area. This may be achieved quite simply by altering the Address constant at $1477 from its current value of $2ØØØ to whatever is required to fit the routines in. For example $4Ø bytes of space between $2ØØØ and $2Ø3F can be made available by changing $1477 to $2Ø and $1478 to $3F.

INITIALISATION

Any initialisation that a specific device may require, must be provided by the driver routines. An ACIA for example must be initialised before it can be used to input or output characters. A suggested method for initialisation is to have the output driver routine detect a paticular character, say $Ø1 and

interpret this as meaning it should initialise the device. An example is provided below of driver routines for an ACIA located in I/O space at $FC to $FF.

```
                    *I/O ROUTINES TO SERVICE ACIA AT FC-FF
                    *
                    STAT    EQU         H'FC'    ACIA STATUS REGISTER
                    DATA    EQU         H'FD'    ACIA DATA REGISTER
                    TBMT    EQU         H'02'    MASK FOR TBMT BIT
                    RDRF    EQU         H'01'    MASK FOR RDRF BIT
                    *
                            ORG         H'2000'
                    *
2000  7710          COUT    PPSL        RS       SELECT UPPER REGISTERS
2002  E401                  COMI,R0     1        INITIALISATION?
2004  980A                  BCFR,EQ     OUT1     OUTPUT CHARACTER
2006  0513          INIT    LODI,R1     H'13'    MASTER RESET
2008  D5FC                  WRTE,R1     STAT
200A  0515                  LODI,R1     H'15'    8 DATA BITS AND 1 STOP BIT
200C  D5FC                  WRTE,R1     STAT
200E  1B08                  BCTR,UN     OUT2
2010  55FC          OUT1    REDE,R1     STAT     GET STATUS
2012  F502                  TMI,R1      TBMT     LAST CHARACTER FINISHED?
2014  987A                  BCFR,EQ     OUT1     LOOP TILL DONE
2016  D4FD                  WRTE,R0     DATA     WRITE THE CHARACTER
2108  7501          OUT2    CPSL        RS       RESET TO LOWER REGISTERS
201A  17                    RETC,UN
                    *
201C  54FC          CHIN    REDE,R0     STAT     GET STATUS
201E  F401                  TMI,R0      RDRF     CHARACTER IN?
2020  987A                  BCFR,EQ     CHIN     LOOP TILL INPUT
2022  54FD                  REDE,R0     DATA     READ THE CHARACTER
2024  17                    RETC,UN
                    *
                    SOP     EQU         $        CURRENT PROGRAM COUNTER
                    *
                            ORG         H'1477'
                    *
1477  2025                  ACON        SOP      START OF PROGRAM POINTER
```

INDEX

MicroWorld BASIC User Registration Form

In order to register your purchase of MicroWorld BASIC please fill out the details required on the card below, and mail it to:-

MicroWorld
BOX No.311
HORNSBY, N.S.W.
2077

This will enable us to keep you informed of any corrections or modifications to MicroWorld BASIC which come to our notice.

Should you become aware of any 'bugs' in MicroWorld BASIC Interpreter, it would be appreciated if you could send full details of the problem, in writing, to the above address. Please supply as many details as possible about the nature of tne problem and include a copy of the actual program which produced the unexpected result.

Please note that this is only for genuine inconsistencies of operation in an unmodified version of BASIC, and does not include 'things I would like this BASIC to do but it doesn't' or problems which occur due to modification of BASIC by the user.

Constructive criticisms or sugestions for improvements to BASIC, in writing, will be welcomed, but no undertaking is given that these will be implemented in a later version of BASIC.

MicroWorld BASIC User Registration Form

NAME_____

ADDRESS_____

_____

PHONE _____ COPY NUMBER _____

PURCHASED FROM _____