Microbee 16K Basic

Users Manual

****

MICROWORLD l6K BASIC FOR THE MICROBEE

USERS MANUAL

****

Welcome and congratulations!
Welcome to the wonderful world of micro computers, now you have computer power at your fingertips.
Congratulations on choosing and using MICROBEE.
You have an Australian computer designed and produced by a skilled and dedicated team of Australians at West Gosford on Australia's central east coast.
Applied Technology Pty; Ltd; is the parent company responsible for the design, development and production of MICROBEE and MICROWORLD BASIC and the selection, editing and publishing of a large, and rapidly increasing range of software programs and manuals.
The immediate acceptance and phenomenal success of the MICROBEE range of Australian personal Computers is due to their outstanding performance for prices lower than foreign imports.
You will be pleased with your MICROBEE'S performance and proud you've bought Australian.

MICROWORLD
P.O. Box 355
HORNSBY 2077
AUSTRALIA

CONTENTS

SECTION1:INTRODUCTION

SECTION 2:A BROAD OVERVIEW.

SECTION 3:MICROWORLD BASIC TUTORIAL

SECTION 4:PROGRAMMING.

Section 1.

Introduction

SECTION 1: INTRODUCTION

1.1 SETTING UP YOUR MICROBEE SYSTEM

1. Plug the 5 pin DIN socket at the rear of the MICROBEE.

2. Connect the video cable to the monitor or modified TV.

3. To the tape recorder plug the BLUE lead into the AUX/MIC and the RED plug to the EAR/MONITOR.

4. Hook up the POWRE PLUG, MONITOR and TAPE RECORDER to the power source.

5. Switch on the power and turn on the monitor. Adjust the monitor (referred to as VDU - for Visual Display Unit) for contrast and brightness, and turn the volume control to its lowest setting.

6. If your Tape Recorder has a Tone Control set it to maximum tone. Set the Volume Control to about 7 or 8. This may need some initial adjustment when you are loading tapes.

[Diagram here.]

1.2 USING THIS MANUAL:

This manual is divided into six sections.

If you are already familiar with this BASIC go straight to SECTION 5 which tabulates in summary form, the STATEMENT and COMMAND DESCRIPTIONS and the FUNCTIONS together with the correct syntax.

If you are not sure, then start at SECTION 2 and work through the examples in this general description. Specfic operations are described in SECTIONS 3 and 4. These should provide useful insight into the various subtle features that are too laborious to describe but become obvious through experiment.

SECTION 6 describes typical examples of commonly used subroutines in MICROWORLD BASIC. These should provide considerable stimulation to developing your own programs under this powerful BASIC.

It is highly recommended that you skim through this manual now to get an overall idea of everything that is detailed. Every attempt has been made to develop the concepts in a logical sequence, however sometimes topics have been introduced out of sequence to help explain a particular point. Make use of the INDEX 8.1 to clarify any particular subject matter you may require.

No doubt you will find yourself asking questions like "I wonder what would happen if....?" or "Will this routine work?". Don’t look for the answer in fine print in this manual; try it for yourself on your computer. The magic of this BASIC is the interaction between you and the BASIC using the error reporting. With a little experimenting you will become quite proficient and capable of adapting MICROWORLD LEVEL II BASIC to any application you require!

1.3 ABBREVIATIONS USED IN THIS MANUAL:

char	ASCII character
int-var	integer variable
real-var	real variable
str-var	string variable
var	general variable (one of above)
line-no	line number
int	integer number
rel-op	relational operator
str-exp	string expression
int-exp	integer expression
real-exp	real expression
exp	general expression (one of above)
[]	ASCII space character
{}	optional specification
<cr>	carriage return
^C	CONTROL key and C together

1.4 WHAT THIS BASIC IS:

This BASIC interpreter has been written for the Z80 microprocessor and has been written to conform as close as possible to the proposed ANSI standard BASIC. As you may already realise there are many "dialects" of BASIC and as such you may have to modify programs from various sources to run them under this particular version of BASIC. Despite the goal of keeping this BASIC as standard as possible it is necessary to detail some differences which set this BASIC apart from others you may be familiar with.

The first and probably the most significant difference is the treatment of real and integer numbers and expressions. As you may realise it is possible to express quantities in integer (i.e "whole number") format and real (or "floating point") format. Storage of the wide range of possible values for each format can consume considerable memory and consequently various techniques are used to anticipate the memory required. Most other BASICS have to "parse" or specially look and process each line to account for mixes of floating point and integer values to optimise the use of memory. The penalty for this is considerable loss of speed.

MICROWORLD BASIC however uses a clever means of circumventing this problem by asking the programmer to decide which format he requires. This is quite logical for many applications require only integer calculations, and these can be performed at maximum speed. Other calculations will need floating point mode and again the programmer has the choice and can select from 4 to

14 place precision depending on his requirements.
The penalty is that you can not implicitly mix the two formats in the same expression and we will discuss this shortly.

MICROWORLD BASIC restricts you to use variable types to designate whether each is floating point or real. Inteqer variables take the form AI B, R, Z, and real variables ˜ust be of the form WI, W7, V0, G3 etc. This means it is not possible to use free form naming of variables such as words or combinations of numbers and letters as variable names. Not much of a penalty really...but them's the rules!!

To adapt most programs to MICROWORLD BASIC, all you need to change is the names of real variables from Q, W, E, R, T, Y, etc to Q0, WI, El, R3, T0, Y7 and make sure you don't mix integers 3nd real expressions. Never fear....if you do make a mistake the comprehensive error reporting in MICROWORLD BASIC will show you EXACTLY where you made the error.

One of the major features of MICROWORLD LEVEL II BASIC is the comprehensive error reporting. Some other BASICS require the programmer to to decode cryptic messages such as uSN ERROR LINE 3300" and usually you waste considerable time in referring to a list of codes. Not so with MICROWORLD BASIC! If you have made QOY error in your program that is not acceptable to BASIC, the MICROWORLD BASIC error report routine stops at the offending line, positions the cursor over the first place where it found the error, identifies the type of error and then prints out an error message in full. An example is the best way of illustrating the point.

```
2990 PRINT "HELLO"
3000 NEXT "HELLO"
     produces when run
Illegal variable error in line 3000
03000 NEXT "HELLO"
           ^ cursor points here
```

Other differences might arise because of the differences in memory automatically set aside for arrays, strings etc. The best policy is to set up sufficient string space and DIMension all arrays before running any program.

No doubt if you are an experienced programmer you will notice several other differences but you will also realise that it is usually a very simple matter to write simple routines to "get around" any command, function or statement requirement.

Section 2.

A Broad Overview

SECTION 2: A BROAD OVERVIEW

2.1 SYSTEM CONTROLS:

This section is just a small summary of some of the features of hardware on the MICROBEE which affect simple operation of the BASIC.

POWERING UP

To start the MICROBEE for the first time, simply apply power, and MICROBEE will respond with the sign on heading.

If the MICROBEE was already "initialised" and the backup memory battery is installed, the computer will only do a "WARM" start. This means that all programs and variables that were in the computer before you switched it off will still be there when you switch it back on again.

THE KEYBOARD:

RESET

The RESET key is located on the bottom right hand corner of the keyboard. The use of this key will NOT destroy programs and variables currently in the machine, but will simply return control from anywhere to the command level after clearing the screen and giving the "Ready" prompt.

To avoid accidental operation of this key, there is a time delay of approximately one second before anything happens. If the key is released within this time, the reset will not take place.

In general, if you desire to regain control of the MICROBEE, try the the break key first and THEN the RESET key only if break has no effect (as in most machine language programs). RESET can be used if for example you wish to abort a SAVE command, or if you type LOAD by accident (in such a case, the original program will still be intact).

Do not use the RESET key during "critical" operations such as when a line is being inserted at the beginning of a large program, or during a renumber, or when a load is half way through.

If it is desired to totally clear the MICROBEE, the "esc" key must be held down while the RESET key is pressed. This will clear the machine entirely and allow recovery from situations such as when critical scratchpads have been poked, or in case a machine language program bombs out (crashes).

To perform this operation, first hold down the "esc" key (top left hand key), and then without letting esc go, press the RESET key for about a second. When a beep is heard and the sign on heading appears, release the esc key. (If you think you might have held down the esc key for too long, just press BREAK to tell the MICROBEE to ignore that line which may have esc characters in

it.)

ABORTING A BASIC PROGRAM NORMALLY

Sometimes it is necessary to abort a running BASIC program, for example to break a loop. To do this you need to press the BREAK key or alternatively, the CONTROL (CTRL) key and C at the same time (Hold down the CONTROL key first, press the C key momentarily, then release the CONTROL key).

This break will stop the operation and return control to the program entry mode.

If for any reason, you just want to hold up a listing or actual proqram execution for a time, and resume exactly where it left off Just press CONTROL S (^S) in the same way as descibed above for ^C. CONTROL S (^S) will temporarily FREEZE the MICROBEE and hitting any key (including ^S) will enable it to RESUME execution from the point at which it was stopped.

SCREEN POSITIONING ADJUSTMENTS

Although the screen display of the MICROBEE is set up for the "average" monitor (converted T.V. set or character display), in some cases centering of the screen is not within the range of the vertical and horizontal hold controls of the monitor. For such cases, the following key sequences will move the entire frame so as to correct for this.

esc,A	(first press the esc key just like any other key, then the A key in the same way, without SHIFT). Move the screen one character position to the left.
esc,S	Move the screen one character position to the right.
esc,W	Move the screen up one line.
esc,Z	Move the screen down one line.

So that you can see where the edges of the screen are, type a full line of characters first, before trying to centre the screen left to right.

It is not necessary to remember which key does what, just notice that the four keys A,S,W and Z form a diamond shape giving the four directions (part of this diamond is also used for horizontal motion in the "EDIT" mode).

```
     W

 A   *   S

     Z
```

When you have finished with these "escape codes", press the "break" key to tell the MICROBEE not to try and interpret these as a BASIC statement, otherwise an error message will be given

when you press the return key.

For example, suppose you had trouble reading the top line of your display; type ESC,Z,BRK and continue programming.

These screen positioning changes stay in force when you switch the power off as long as you are using the backup battery. They are however cleared when the COLD start is given (holding down escape and pressing RESET).

For the adventurous (and careful) programmer, these escape codes can also be used in a print statement (to print an escape without it appearing in the listing and disturbing your screen, use the PRINT CHR$(27); construction.)

GENERAL KEYBOARD OPERATION

The MICROBEE keyboard consists of 60 keys which perform both character entry and control functions.

All keys except the RESET, SHIFT, CONTROL and LOCK work in the following manner; when the key is initially depressed, one character is generated. While the key is kept held down, the MICROBEE waits for one second. If the key is still held down by this time the MICROBEE will start producing that character at the rate of approximately l0 characters per second until the key is finally released.

This easy method of producing many of one character is called "auto-repeat", and is a feature usually only found on expensive terminals.

When an alphabetic key is pressed on the MICROBEE keyboard, the code generated is usually lower case (unless the alpha-lock has been activated). This means that to get an upper case character, you first hold down the shift key and then press the desired alphabetical key.

Lower case is fully provided for in MICROBEE BASIC, all program entry and editing may be done in lower case. The BASIC converts all BASIC keywords (such as PRINT, LET) and variable names to upper case, while retaining the entered case for strings (as in PRINT "Hello, how are YOU"). REMark statements also retain the case of input.

To explain this through examples, suppose the following was typed in (don’t worry about what the commands actually do yet).

```
10 print "BASIC is an Acronym"
29 rem That was a print statement
39 end
```

The MICROBEE will convert it to this internally:

```
00010 PRINT "BASIC is an Acronym"
00020 REM That was a print statement
00030 END
```

The shift key also allows access to the punctuation characters found printed on the top of some of the keys. For

example, if the "1" key is pressed by itself, the numeral 1 is generated, but if the shift key is held down first, the exclamation mark is generated.

The CONTROL (marked as CTRL) key allows a third meaning for some of the keys on the keyboard. As mentioned before, ^S (abbreviation for CONTROL S) provides a means of pausing the MICROBEE's operation for a while. Other CONTROL characters may also be of interest:

^G sounds a beep from the speaker (bells were used in times of old).
^A moves cursor to the left when in "EDIT" mode.
^S moves cursor to the right when in "EDIT" mode as well as pausing the computer when RUNning a program or LISTing a program.
^W moves the cursor to the next character after the next patch of spaces when in "EDIT" mode.

Most of the control characters used however, have separate keys. Such keys are

RETURN or ^M Enter the line of characters typed so far as something to be acted upon (a new program line to be inserted, or some immediate command to act upon.)

DEL Normally, delete the character to the left of the cursor and then move the cursor to the left. In "EDIT" mode, this key deletes the character which the cursor is sitting on, and moves the rest of the line back to the left.

BACKSPACE or ^H This key normally performs the same function as the delete key, but differs merely in that the character that is being deleted is not scrubbed from the screen.

BREAK Used to regain keyboard control when a BASIC program is running.

ESC or ^[This key is used as described above to clear the MICROBEE out totally and to reposition the screen in conjunction with the a,s,w,z keys

LINE FEED or ^J This character moves the cursor down the screen one line (not usually used in BASIC).

TAB or ^I Tabulate function (not usually used in BASIC), but used by such things as EDITOR-ASSEMBLER.

The alpha-lock, or "LOCK" key is a special keyboard function key which generates no codes, but merely swaps the keyboard between two states. Normally, the keyboard makes lower-case characters when alphabetical keys are pressed, and upper case when the shift key is used. It is sometimes useful, however, to make upper case characters normally, and lower case when the shift key is used.

The LOCK key toggles the MICROBEE keyboard between these two modes of operation.

2.2 CONSTANTS, VARIABLES AND EXPRESSIONS:

Before you can understand the rest of this manual you will need to have a clear understanding of each of these fundamental concepts From here on we will be using the terms CONSTANT, VARIABLE and EXPRESSION freely and it will be assumed that the reader Knows the relationship of each 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 are all NUMERIC CONSTANTS. The "E" in the 1.98E-6 is a shorthand notation to indicate scientific notation, i.e. 1.98 X 10 to the power of -6.

Alternatively, a constant may be a STRING CONSTANT, in which it will consist of a "string" or sequence of any printable characters.

For example: "MICROWORLD" "HELLO 96, HOW ARE YOU" are STRING CONSTANTS.

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

Each of the lines below contains a CONSTANT.

```
10 LET Al=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. This NAME must be selected to indicate if the VARIABLE is an INTEGER or a REAL (floating point) NUMBER. The following are some examples of valid variable names. See below for more information about naming variables.

For example: A, T, Z, X are valid INTEGER NUMERIC VARIABLES

A1, T0, Z4, Xl are valid REAL NUMERIC VARIABLES
Al$, R4$, H0$ are valid STRING VARIABLES

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 or STRING OPERATORS
FUNCTIONS

For example: typical EXPRESSIONS are

```
(A1+4)*B1
INT(RND*6)+1
LEN ("ABCDE")
"THIS IS "+" AN EXPRESSION"
```

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 (A1$+"FRED"+B2$)
SIN(360*E2+0.1)
POINT ( INT(CI+V4),INT(C2+5*M0) )
```

BASIC will evaluate any EXPRESSION first, according to its priority rules. These are treated shortly 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 an error message.

```
For example:   10 LET A1$=B1$+A1        INVALID
               20 A1=SIN(A1$+1)         INVALID
```

HOW COMPLICATED CAN AN EXPRESSION BE?

The length of one line of BASIC code is limited to 184 characters, the length of the input buffer, so that an expression can never be longer than one line. If more than 184 characters are typed on the one line the last characters are not echoed, and will not form part of the line.

More importantly, the intent of a long expression can be easily obscured by its complexity. It is better programming practice to break a long expression into a couple of short expressions. Short lines are also easier to correct when you make a typing error. The long expression in line 10 of this example calculates the resistance of three resistors in parallel.

```
For example:  5 REM R0 = EFFECTIVE RESISTANCE OF R1,R2,R3
              10 R0=1/((1/R1)+(1/R2)+(1/R3))
```

This can be more clearly written in four lines:

```
5 REM C0 = CONDUCTANCE
10 C1=1/R1
20 C2=1/R2
30 C3=1/R3
40 C0=C1+C2+C3
50 R0=1/C0
```

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

2.3 ENTERING A PROGRAM

The first two COMMANDS with which to become familiar are
NEW LIST

NEW tells BASIC that it should delete from memory all traces of any previous program and that you are about to enter a fresh program. MICROWORLD BASIC will clear the program buffer and prompt with

```
>
```

LIST will list the program you have entered. LISTing is discussed in greater detail later.

YOUR FIRST PROGRAM

Suppose this is your first time and there is no existing program in memory. After typing NEW, type a carriage return <cr>. Every time you want BASIC to act upon a line of text or commands you have just typed in, press the CARRIAGE RETURN <cr>. Nothing

will happen until you do. If you make a mistake while typing in a line, you can use the BACKSPACE key to backspace along the line to the mistake, then simply type the correct letters over the top. After you have pressed <cr> however you will have to use the EDIT mode described shortly.

2.4 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 each line exactly as it appears and 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 N1$
30 PRINT "HELLO ";
40 PRINT N1$
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>

BASIC will start executing the instructions contained in the program beginning at the lowest line number, in this case 10, and working up, in order of line number.

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 VDU screen. 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 have pressed <cr> to tell BASIC 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 N1$.

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 is enclosed in quotes. The semicolon at the end of the line tells BASIC not to 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 N1$, 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 the line to be stored in memory" rather than acted on immediately. If the line does NOT begin with a number then BASIC assumes that the line is a command like LIST, RUN or some other instruction and that you wish to act on that instruction immediatley. Most BASIC statements can be executed directly. This is called IMMEDIATE MODE OPERATION and is described shortly.

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 (GOTO) in the program interrupts this sequence. If the program does contain branch instructions, then the line numbers provide a branch address for the branch to go to.

Thirdly, if you want to alter the program, you change a line by either retyping a new line using the same line number or using the EDIT mode. Alternatively you can insert a new line number between existing line numbers. That is why line numbers in the sample program increment by 10, to allow plenty of space for extra lines to be added. The maximum number of line numbers allowed is 65534, so that 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 the KEYWORD. A KEYWORD is analogous to the verb in an English sentence and a STATEMENT is analogous to a complete sentence. Each line can contain more than one statement if it is separated by a colon ':'.

MULTIPLE STATEMENT LINES

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

```
10 PRINT "WHAT IS YOUR NAME ";:INPUT N1$: PRINT "HELLO";N1$: 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 line statement is 184 characters, since this is the length of the input buffer.

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 one line. However if the line contains an IF...THEN statement, the statement on the line after the THEN will only be executed if the IF...THEN is TRUE. If it is FALSE, the program either executes statements after the ELSE keyword or if there is no ELSE, moves to the next line number and will ignore all statements after the THEN. This technique is used to avoid using the GOTO instruction since it allows more than one instruction to be executed if the test fs TRUE.

AUTO LINE NUMBERING

A feature of MICROWORLD LEVEL II BASIC is the AUTO line numbering facility. This saves considerable time and effort and leaves you free to concentrate on the programming. To engage AUTO simply type 'AUTO 10,10'. This tells the BASIC you want to enter a program starting at line number 10 and stepping ahead in multiples of 10 lines at a time. You can select any combination such as 'AUTO 100,5' or, if you wish you can type 'AUTO <cr>' and the BASIC will start at line number 100 and proceed in steps of 10 lines. To qet out of the AUTO mode simply press <cr> twice and the line numbering returns to the manual mode. Refer to the command list for more details.

LISTING

To LIST the program you have entered use the LIST command. 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 of the lines between line 20 and line 30.

NOTE: LLIST is the same as LIST except the output goes to the printer stream instead of the terminal.

END

Line 60 is an END statement. It tells the 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 REM on a line is skipped over by the BASIC program, including multiple statements! REMarks do take up memory and also take time to enter. A REMark may not have any other statements following it since BASIC considers anything that 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..

2.5 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 <cr>. 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" <cr>
```

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         16        216
     7         49        343
     8         64        512
     9         81        729
     10        100       1000
```

All this was done with one line of BASIC code. The immediate mode allows your computer to be used as a powerful calculator merely by typing 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.

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 values and all arrays will remain set up so that the contents of these may be inspected by 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 typing CONT to CONTINUE the program.

END, however, often signifies the physical end of the program as well as terminating the execution of the program, although there is no compulsion for END to be the last statement of the program.

FRE

If you need to know how much memory you have left for program and variable storage you just type PRINT FRE(0) and BASIC will return a number representing the available memory.

SAVE

The SAVE command can be used to save the BASIC source file currently in memory on cassette tape. One correct format is SAVE "file n,me" where "file name" can be up to 6 characters long. (See Section 4: Statements and Commands).

LOAD

The LOAD will load the SAVEd program back into memory. The simplest format is LOAD <cr> .

2.6 VARIABLES AND ARRAYS

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. MICROWORLD BASIC supports 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 may be either INTEGER or REAL. If INTEGER the number must lie between -32768 to +32767. If REAL (or 'FLOATING POINT') it must lie between 1.0000000..*10^-64 to 9.999999..*10^+62. The number of significant digits is user selectable between 4 and 14.

A STRING VARIABLE stores a string of characters and treats them as one entity. An example of a STRING would be a line of text. A STRING is of variable length, only the number of characters actually used are stored. The maximum length of a STRING variable is 255 characters.

VARIABLE NAMES

The type of 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 to 7 inclusive. An INTEGER NUMERIC VARIABLE (i.e a WHOLE number like number of oranges) is represented by a single alphabetic letter such as A, M, X, Z. A REAL NUMERIC VARIABLE must be represented by an alphabetic letter followed by a digit between 0 and 7 such as Q7, A1, Z0, X3. This is an important difference between MICROWORLD BASIC and other versions of BASIC. NOTE also that you can't mix INTEGER and REAL VARIABLES in the same expressions or a MIXED MODE ERROR will result.

A STRING VARIABLE name consists of a single letter A to Z, followed by a number 0 to 7, followed by '$'. For example J0$, X7$ are valid names for STRING VARIABLES.

Examples of VALID VARIABLE NAMES: A, W1, B6$
INVALID VARIABLE NAMES: AD, T$, QW$, Z9

ARRAY VARIABLES

Often it is useful to group variables together into an ARRAY. This enables you to group these similar variables using the same variable name, but be able to access each variable indivIdually by means of an index.

An ARRAY VARIABLE 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 can 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 Q1$(4) and look like this:-

INDEX	CONTENTS
Q1$(1)	"HARRIS"
Q1$(2)	"STARR"
Q1$(3)	"HILL"
Q1$(4)	"MILLS"

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

PRINT Q1$(2) and "STARR" would be printed out.

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. To tell the BASIC how much space to leave it is necessary to DIMension arrays before using them. In our example above would require a statement:

```
10 DIM Q1(4)
```

This will reserve space for an array of four variables which can from then on be assigned as being a string or a real number. In this case, we assigned the variables as Q1$(1) (for example), so they are treated as strings.

ARRAYS can be either single dimension as above or may have more dimensions. In this latter case the DIM statement is as follows:

```
10 DIM B1(10,20)
```

This reserves space for an array of REAL NUMERIC or STRING VARIABLES up to 10 rows down and 20 columns. To access an element of the array you type

```
10 PRINT B1(5,18) or
10 PRINT B1$(5,18)
```

Depending upon whether you had assigned it as a string or a real.

This would produce the value of the variable stored in the 5th row and the 18th column across.

Note that arrays are allowed to have their first index as a 0 ; but they may be treated as starting at 1 if desired.

NOTE: Restriction on Variable Names

MICROWORLD BASIC restricts the use of variable names of the same letter. You cannot use Al$ and A1 in the same program because the BASIC will not distinguish the string variable from the real numeric variable. The type of variable that A1/A1$ is at any time is determined by the last assignment made to that variable.

Section 3.

Microworld Basic Tutorial

SECTION:3

A TUTORIAL IN MICROWORLD BASIC

This section of the manual has been written for rank beginners using a new practical approach which will be effective in introducing MICROWORLD BASIC as quickly as possible.

The MICROBEE is a friendly computer which we will use to show you how to use MICROWORLD BASIC. If you require specific descriptions of commands, statements and functions under this powerful BASIC refer to Section:5. The following examples should provide useful reference when you are writing programs and need to cross check a particular point. This method is particularly effective if you are familiar with another version of BASIC as the exercises will serve to highlight differences and subtle points that are often buried in the text of other user's manuals.

3:1 INTRODUCING THE SYSTEM

This is more "wordy" than most because of the necessity to introduce some fundamental concepts before we can proceed to actually use the MICROBEE. It is recommended you complete each of these exercises to familiarise yourself with the MICROBEE and how it works.

CONCEPTS INTRODUCED. When you complete this experiment you should be familiar with the following

The NEW command
SOFT START
HARD START
Repositioning the image on the VDU screen
IMMEDIATE MODE
INDIRECT MODE
ENTERING a line of program
The RUN command
The PRINT command

The MICROBEE in its standard form is supplied with MICROWORLD BASIC stored in permanent memory. When you have set up your MICROBEE 16K or 32K using the directions on the connection diagram and switch it on, the BASIC program runs immediately.

1. TURN ON THE MICROBEE AND PRESS THE RESET KEY

The MICROBEE will respond with

```
Ready
>_
```

This indicates that the BASIC is ready to receive input from the keyboard. MICROWORLD BASIC accepts upper and lower case

characters. However to highlight the steps you carry out in these exercises we will type in UPPER CASE only. To enter anything into the BASIC you just type normally on the keyboard and, when you have finished the word or instruction, press the key marked RETURN. This is often abbreviated to <CR> meaning press CARRIAGE RETURN (from the old typewriter days).

TYPE: NEW <CR> (That is press the key 'N' then 'E' then 'w' and then press the 'RETURN' key)

This initialises the BASIC ready to accept a new program. Because the MICROBEE is equipped with continuous memory, it has bee fitted with two types of RESET facilities. These are called HARD START and SOFT START. A SOFT START resets the program back to the READY >_ mode but retains the program intact in memory.

FOR A SOFT START PRESS THE RESET KEY

and the MICROBEE will respond with a clear VDU screen with

Ready
>_

in the top left hand corner.

A HARD START involves resetting the computer so that all the memory is cleared and any BASIC programs (apart from those in ROM of course) are erased from memory. A HARD START occurs when the MICROBEE is powered up when the memory backup batteries are disconnected. You can also force the MICROBEE to execute a HARD START by holding down the RESET key for longer than 2 seconds and then hold down the ESC key with the other hand and, while still holding down the ESC key release the RESET key. You will know a HARD START has occurred because the internal loudspeaker will sound a short tone and the MICROBEE will sign on to the initial screen message.

Should you need to adjust the position of the screen display on the VDU screen, you should examine the following keys on the keyboard:

ESC

W

A S

Z

ESC S moves the cursor one space to the right
ESC A moves the cursor one space to the left
ESC W moves the cursor up one line
ESC Z moves the cursor down one line

Note that this information is stored in memory in the MICROBEE and is retained during a SOFT START. However, a HARD START will destroy the information and may require you to reposition the screen.

IMMEDIATE MODE

The power of BASIC is illustrated with the following example. Try it for yourself by typing on the keyboard directly. Remeber that <CR) is our shorthand for 'PRESS THE RETURN KEY' and this is used to tell the BASIC to enter the line of program.

TYPE: P R I N T 4 + 2 <CR> and the computer will respond with

6.

You have now used the BASIC to calculate the sum of 4 and 2. This is called the IMMEDIATE MODE and, as you see, this causes BASIC to evaluate the line of program IMMEDIATELY. Why not try things like PRINT 60 + 20 - 5 <CR). (Notice that the '-' sign and the '=' operator are on the same key.) Should you wish to multiply use the '*' key and divide use the n/n key. A little experimenting will show how these operate.

INDIRECT MODE

Most BASIC programs are written in the INDIRECT MODE, using LINE NUMBERS. The INDIRECT MODE is the NON-IMMEDIATE or deferred execution mode. The program is not executed until the RUN is given.

TYPE:
NEW <CR) to clear out any program in memory
TYPE:
100 PRINT 4 + 2 <CR)

The command)- waits for more input or a command.

TYPE:
RUN <CR>

and the computer gives the output 6.

To program a message, the print command is used. The message must be written between quotation marks (" "). Add these lines to the program:-

TYPE:

```
110 PRINT "WELCOME TO THE MICROBEE." <CR>
120 PRINT "PROGRAMMING IS FUN. " <CR>
```

TYPE:

```
RUN <CR>
```

and the computer will respond with

```
6.
WELCOME TO THE MICROBEE
PROGRAMMING IS FUN
```

The use of LINE NUMBERS in the INDIRECT MODE has retained the program in memory until the command RUN has been given. To check that the program is still in memory

TYPE:

```
LIST <CR>
```

and the MICROBEE will display

```
00100 PRINT 4 + 2
00110 PRINT "WELCOME TO THE MICROBEE."
00120 PRINT "PROGRAMMING IS FUN."
```

NOTE: You must use '0' for zero, not the letter 'O'.

EXERCISE:

To check that you have grasped the fundamental concepts so far go back to the INDIRECT MODE. Press the RESET key to perform a soft start and do this:

TYPE:

```
NEW <CR>
```

TYPE:

```
100 PRINT 100 + 10 <CR>
```

Now test yourself with the following questions.

1. What will happen when you type RUN <CR>?
2. What will happen when you type LIST <CR>?
3. What is the difference between the immediate mode and the indirect mode?
4. If you type NEW <CR> and LIST <CR> what should happen?

Check yourself against the MICROBEE.

If you were a beginner at the start of this exercise you have lost that status and are well on the way to becoming a a master at MICROWORLD BASIC!

To further improve your skills, repeat this exercise using upper and lower case characters and also inserting spaces (by using the space bar) liberally throughout the program. Notice that BASIC ignores spaces and whether you have typed in upper or lower case.

Now is the time to test three other features of the MICROBEE keyboard.

1. press down any white function key such as 'Z' and watch what happens. Notice that it AUTO REPEATS after a short delay.

2. To 'RUB OUT' the characters typed in press the DEL key and watch what happens.

3. Press the LOCK key once and you will notice that any character typed will be in UPPER CASE even though the SHIFT key is not pressed. Press the LOCK key once again to return to normal operation. It would be a good idea at this point to experiment for yourself with the other keys to check the function of each.

3.2 LINE NUMBERING

CONCEPTS: AUTOmatic insertion of line numbers
Manual insertion and deletion
RENumbering lines

The INDIRECT MODE is the usual mode used for BASIC programs. It involves inserting a line number for each program step. BASIC executes the program starting from the lowest numbered line and proceeding line by line until it has completed the highest numbered line. Note that the line numbers do not have to be sequential. It is a good idea to leave spare numbers between program line numbers so that other program steps can be added at a later stage.

TYPE:

```
NEW <CR>
```

and enter the following program:

TYPE:

```
100 PRINT 4 + 2 <CR>
110 PRINT 4 + 3 <CR>
120 PRINT 4 + 4 <CR>
```

TYPE:

RUN <CR>

and the computer will respond with

6.
7.
8.

The program performed line 100 first, then 110 and finally line 120. To test this try the following:

TYPE:

105 PRINT 1 + 1 <CR> and then type LIST <CR>. Notice that the new program line was automatically inserted by the BASIC between lines 100 and 110. Test it for yourself by typing RUN <CR> and check that the computer outputs 6, 2, 7, 8 indicating that line 105 was indeed executed ahead of 110.

AUTOMATIC LINE NUMBERING

MICROWORLD BASIC allows you to insert line numbers automatically to save the time and effort involved in inserting line numbers for every program step. Try the following:

TYPE:

NEW <CR>

TYPE:

AUTO <CR>

and the BASIC will respond with

00100 _

Now enter one line of program (e.g. PRINT 4 + 2 <CR> and notice that it is entered after the line number. When you press <CR> the computer responds with

00110 _

You should enter another line finishing with <CR> and notice that the line is entered and the next line number is incremented.

When you have finished inserting your program press <CR> twice and the BASIC will exit from the AUTO insert mode.

Notice that the AUTO command only operates in the immediate mode and defaults (computer jargon for ’goes to automatically’) to a format starting at line number of 100 and increments at 10 lines at a time. If you type AUTO after you have already entered part of your program the insertion will start a new line after the

program lines already in memory. If you want to enter lines starting at a llne number other than 100 and with an increment other than 10 you must specify both with the AUTO command.

TYPE:

AUTO 1000,25 <CR>

will start line insertion at line 1000 and increment at 25 lines at a time.

you should try this for yourself. What does AUTO 1000 <CR> do?

DELETING LINES IN BASIC

We can insert lines in a BASIC program. How do we eliminate them if we don't need them?

A. To eliminate one line from an existing BASIC program just type the line number and then enter <CR>. Try it for yourself.

B. To delete a specific line (e.g 110) you can type DELETE 110 <CR>. To delete a larger block of lines type

DELETE 100,150 <CR>
where 100 is the first line to be deleted and 150 is the last.

TYPE:

NEW <CR>

AUTO 500,20 <CR>

inserts lines starting from 500 and incrementing in steps of 20

500 PRINT 4 + 2 <CR>
520 PRINT "EXERCISE 1" <CR>
540 PRINT "EXERCISE 2" <CR>
560 PRINT "EXERCISE 3" <CR>
580 PRINT 3 + 4 <CR>
600 <CR>

TYPE:

LIST <CR>

and notice that the BASIC lists all the lines. Now type

TYPE:

DELETE 520,560 <CR>

and then TYPE:

LIST <CR>

now notice that lines 520 to 560 have been deleted leaving the remainder intact.

RENUMBERING LINES

A useful feature of MICROWORLD BASIC is the ability to renumber lines. This is useful if you have run out of space between lines and need to insert another program line. Consider the following example:

```
100 PRINT 4 + 2
101 PRINT 4 + 4
102 PRINT 4 + 5
103 PRINT 4 + 6
```

You can insert this for yourself with the commands NEW <CR> and then AUTO 100,1 <CR>. Now suppose you want to insert the step PRINT 4 + 3 after line 100 you will notice that line 101 is already allocated. Type LIST <CR> to verify this.

TYPE:

RENUM <CR>

this will renumber all lines starting at 100 and incrementing in steps of 10 lines. To confirm this

TYPE:

LIST <CR>

and now insert line 105 as follows

TYPE:

105 PRINT 4 + 3 <CR>

Renumber is formatted similarly to AUTO. Typing RENUM 1000,10 will renumber the entire file so that the first line number will be 1000 and incrementing in steps of 10. The command RENUM 1000,10,130 will renumber part of the program starting at line 130 and renumbers line 130 to 1000 and increments the rest of the lines by 10.

By now you should be able to INSERT line numbers manually and AUTOmatically, DELETE and RENUMber lines. These skills are best practised with more meaningful programs which you will be able to write soon.

3.3 CONSTANTS AND VARIABLES IN MICROWORLD BASIC

CONCEPTS

NUMERIC and STRING VARIABLES
INTEGERS
REAL NUMBERS
The INPUT command

Up to this point we have only used values in our programs which are constants. Specifically these have been numbers which do not change throughout the program and these are called numeric constants. Constants are just numbers and can be represented in MICROWORLD BASIC as INTEGERS (whole numbers only) or FLOATING POINT (REAL numbers like 3.2345 or -2.34).

MICROWORLD BASIC can also deal with 'words' and these are called 'STRINGS'. (See example 1 for a full explaination.) It is possible for a program to contain a string constant as well. One example might be:

```
100 PRINT "HELLO"
```

here the word 'hello' is called a string constant. Notice that we enclose strings inside quotation marks to enable BASIC to recognize them. A string constant in MICROWORLD BASIC is a sequence of up to 180 alphanumeric characters enclosed in quotation marks (e.g. "HELLO").

MICROWORLD BASIC also uses variables. These can also be REAL, INTEGER or STRING variables. These have particular names so that BASIC can treat each separately.

TYPE	NAME	RANGE
REAL	An (where n= 0 to 7)	+/- 9.99999999999999 E+62 to +/- 9.99999999999999 E-63
INTEGER	A	-32767 to + 32767
STRING	An$	0 to 180 characters

Where 'A' is any letter of the alphabet and 'n' is a number from o to 7. Real variable names and string variable names should not use the same letter and digit within the same program or errors will result. String variables MUST use An$ construction and NOT A$ which is invalid. Note that MICROWORLD BASIC will not allow you to mix REAL and INTEGER numbers on the same program line.

Try a small program. Type NEW <CR> then AUTO <CR> and insert the following:

```
100 PRINT "What is your name "; REM note this semicolon!!
```

```
110 INPUT N1$
120 PRINT "Glad to meet you  ";N1$
```

This little program introduces a new command, INPUT. INPUT requests data from the keyboard and then assigns it to the string variable which is then printed out in line 120. Try the program for for yourself and enter <CR> when you have finished typing your name. By the way, the semicolon (;) at the end of line 100 means that as you type your name it stays on the same line. Notice that the question mark (?) is generated automatically.

TYPE:

```
RUN <CR>
```

and the computer will respond with

What is your name?

(You type in your name here followed by the <CR> to return to BASIC) and the computer will print

Glad to meet you (your name!)

Modify the program to make it more interesting. Retype line 120 as follows:

```
TYPE:     120 PRINT "How old are you ";N1$;<CR>
TYPE:     130 INPUT A <CR>
TYPE:    140 PRINT "Are you really ";A;"years old";N1$;"?"<CR>
TYPE:     LIST <CR>
```

TYPE: RUN <CR> and answer the questions. Remember to use the <CR> to enter each answer. Note that the second question uses INPUT A which is a numeric integer variable so be sure to enter your age as a whole number of years!

EXERCISE: Using the above program write down an example of each of the following:

(a) string constant
(b) string variable
(c) a numeric variable

NOTE: By now you should be gaining confidence in entering programs under BASIC. You should be able to use the RESET key for a SOFT RESET, type NEW to erase an old program, use the AUTO command and also the RENUMber command. You have already encountered BASIC commands such as LIST, RUN, INPUT, and PRINT. If you are not sure of any particular point at this stage it might be a good idea to read through the particular exercise, trying variations on programs and referring to the detailed section later in this manual.

Once you have mastered these fundamental concepts you will be ready to proceed to the next set of exercises.

3.4 AND LOADING PROGRAMS ON CASSETTE TAPE

Connect a low cost tape recorder (preferably mains operated) to the MICROBEE using the coloured 3.5 mm leads-provided. Connect the RED plug to the EARPHONE and the BLUE plug to the EARPHONE output on the cassette recorder. Plug the recorder into the 240v mains, insert a blank cassette (rewound of course) and you are ready to proceed with this exercise.

CONCEPTS

SAVE	saving at 300 BAUD
SAVE F	saving at 1200 BAUD
LOAD	loads any program
File types	
BAD LOAD	
LOAD U	
LOAD?	

SAVING PROGRAMS

MICROWORLD BASIC has built-in commands to enable you to save a BASIC program. Even while you are aeveloping a long program it is a good idea to save it just in case something goes wrong. Supploe we have the following program in memory:

```
100 REM this program prints a table of powers
110 FOR A1=l to 10
120 PRINT A1, A1^2, A1^3, A1^4
130 NEXT A1
```

We can save this program to tape as follows:

1. Connect the tape recorder and switch it on.

2. Insert a cassette (if you have not already done so) and wind it on until it is past the leader (clear section of tape).

3. Press the RECORD and PLAY buttons on the cassette player as you would when recording normally and

4. Type:

   SAVE "TEST1" <CR>

5. Wait a short time and you will hear a ’beep’ indicating that the program has been saved.

Note that this process has SAVED a program called TEST1 on the cassette tape presently in the recorder. The SAVE was at 300 BAUD which is adequate for most applications.

To SAVE at 1200 BAUD you type

SAVE F "TEST1" <CR>

LOADING A TAPE

To reload the test program or the demonstration cassette supplied, type NEW <CR> and then follow the procedure detailed below:

1. Rewind the tape containing the program.

2. TYPE:

LOAD <CR>

3. Press the PLAY button on the tape recorder.

The screen display on the MICROBEE should respond with the program name (e.g. NIM B*) (note that the '*' flashes as loading is taking place and the 'B' means that you are loading a BASIC file.)

When your program has been loaded, you will hear a 'BEEP' indicating that BASIC is now ready to RUN the program. stop the tape recorder and type RUN <CR>.

When you have finished with the program in memory press SOFT RESET and type LOAD <CR> again. press the PLAY button on the recorder and the process will be repeated.

NOTE: The LOAD command is identical for 300 and 1200 BAUD tapes. The speed is determined by the program on the tape and adjusts automatically.

BAD LOAD

When you command the BASIC to LOAD the program is read from the cassette tape and written into the BASIC file memory. To ensure that errors don't occur, the BASIC performs what is called a CHECK SUM (CRC) error test to verify that the data read from the tape matches that which was recorded beforehand.

If the CRC doesn't match then

BAD LOAD appears and the program does not load into memory. If this error message occurs, rewind the tape and start again. Possibly you should experiment with a different volume control setting (about 7-8 on a scale of 10 should work fine).

LOAD U and LOAD?

TWO further variations of the LOAD command are used under MICROWORLD BASIC.

LOAD U is used to override the checksum error used. with the usual load command. You can use this facility to 'fix' a faulty tape and recover mutilated or otherwise faulty programs. Remember that it is highly likely that the program when loaded under LOAD U will contain errors so you will have to edit carefully before typing RUN.

LOAD ? is a useful feature for checking that a program you have just saved will load without checksum errors. In effect it enables you to run the tape through without actually loading the tape and overriding the program currently in memory.

MACHINE CODE PROGRAMS

Your MICROBEE can run programs written in machine code (the actual language used by the Z80 processor) as well as BASIC. Running machine code programs has the advantage that they run much faster than under BASIC and, in most cases, require far less memory. To load a machine code tape you still type:

LOAD <CR>

but the file type will load as a type 'M' as follows:

file M * i.e. a 'M' appears after the file name in place of the 'B' as with BASIC programs.

Nothing else changes. Note however that there is no corresponding SAVE command for machine code programs unless you use the optional MICROWORLD machine code monitor.

When a machine code program has been loaded DO NOT TYPE RUN. Most machine code programs are intended for AUTO STARTING, however if you press RESET then you will need to type:

EXEC <CR> to start the machine code program.

3.5 EDITING PROGRAMS

CONCEPTS

EDIT MODE
ERROR MESSAGES
AUTO EDIT feature
^A moves cursor to the left by one character
^S moves cursor to the right by one character
^W moves the cursor to the right one word

DELete
GX global search and replace

One of the most powerful features of MICROWORLD BASIC is the ease of editing programs. When you have finished typing a line under BASIC it is enter red into the BASIC memory area the moment the <CR> has been pressed. To alter this line at any later time requires that you enter the EDIT mode.

Try this example to learn about the EDIT feature. TYPE:

NEW <CR> and then

AUTO <CR> and the MICROBEE will respond with

100 _

and you should enter the following line

100 PRINT "COMPUTERS DON'T MAKE MISTEAKS <CR>
110 <CR>

Now look at what we have done! A spelling mistake (misteak???) exists in line 100. To edit the line you type

EDIT 100 <CR>

and the computer will respond with

00100 PRINT "COMPUTERS DON'T MAKE MISTEAKS

notice that the cursor is located under the 'P' in the line. To move the cursor use ^S (CONTROL and S pressed at the same time). Notice that the cursor moves along the line to the right. Try ^A and see the cursor move to the left. Also try ^W to check that the cursor moves to the right one word (or a complete block between spaces) at a time.

With a little practice you should be able to position the cursor below the 'E' in MISTEAKS. Press the DEL key and DELete the 'E'. OBVIOUSLY DEL DELETES THE CHARACTER ABOVE THE CURSOR. Now press <CR> and then type LIST <CR>. The MICROBEE will respond with:

00100 PRINT "COMPUTERS DON'T MAKE MISTAKS

Type EDIT <CR> and the MICROBEE will respond with

00100 PRINT "COMPUTERS DON'T MAKE MISTAKS

Now position the cursor under the second S in MISTAKS and type E. IN THE EDIT MODE CHARACTERS ARE AUTOMATICALLY INSERTED UNDER THE CURSOR, FORCING THE REMAINING CHARACTERS TO THE RIGHT BY ONE CHARACTER SPACE. Press <CR> and you will have the line in memory.

line 100 is intended to instruct BASIC to print the string 'COMPUTERS DON'T MAKE MISTAKES'. You may recall that strings must it enclosed in quotation marks (" ") AT BOTH ENDS. Let's try and RUN this line.

TYPE: RUN <CR>

and the MICROBEE will respond with

```
Missing end quote error in line 00100
00100 PRINT "COMPUTERS DON'T MAKE MISTAKES
```

How we have received an error message from the BASIC. In this case, it has told us what was specifically wrong and in which line number the error was detected. It has also bought line 100 into the edit buffer and typing EDIT <CR> will list this line and enable you move the cursor to the far right end of the line and insert the end quoted (") as required. Now type RUN and notice that the MICROBEE will run the program and print out the required aessage.

AUTO EDIT MODE

If you type AUTO 100 <CR> you will notice that the computer will respond with line 100 on the screen in the EDIT mode. You can correct the errors and when you press <CR> the next line will be displayed. If you had had a problem in the program you can correct it, alternatively you can insert new program lines after the existing program line numbers every time <CR> is pressed.

GX GLOBAL SEARCH AND REPLACE

Sometimes when you are debugging a program it is necessary to search the whole program for the occurence of a particular word, string or variable and replace it with another. In our example let's search for the word 'COMPUTERS' and replace it with 'PEOPLE' as follows. With line 00100 in memory from the earlier part of this exercise.

TYPE: GX/COMPUTER/PEOPLE/ <CR> and the MICROBEE will respond with

00180 PRINT "COMPUTERS DON'T MAKE MISTAKES" with the cursor under the 'S' in COMPUTERS. Press the'.' (full stop) key and the replacement has taken place. pressing any other key would have caused the routine to bypass the first occurence of COMPUTERS and searched for the next. Type LIST <CR> and see what has happened.

EDITING LINE NUMBERS

Using the ^A facility you can also edit a line number. This is a neat method of avoiding retyping lines with similar expressions or that are very similar. Note that when ypou edit and produce a line number the old one remains in memory. The best

approach is to try it for yourself.

3.6 GRAPHICS (LOW AND HIGH RESOLUTION)

CONCEPTS:

- LORES
- HIRES
- SET, RESET
- SETH, RESETH
- PLOT, PLOT I, PLOT R
- USED

The MICROBEE has a very flexible graphics capability. The high resolution (HIRES) graphics has a resolution of of 512 dots by 256 dots and the low resolution (LORES) mode has 128 by 48 dots. The graphics are generated using a programmable CHARACTER GENERATOR since this technique makes most efficient use of memory without losing valuable program space as in other computer systems.

In the HIRES mode the actual PCG (programmabel character generator) characters are used to develop the graphics. Because these are limited to 128 different characters you may have to take care with complex graphics using the total screen area. In the LORES mode no such limits are imposed and the screen graphics are similar to those used by the TANDY in the TRS80 computer.

Type in the following program:

```
100 CLS: LORES
110 X=INT(RND*128):Y=INT(RND*48)
120 SET X,Y
130 GOTO 110
```

The program proceeds as follows:

Line 100 clears the screen and selects the LORES mode (note that we can put two or more commands on a line if they are separated by a colon).

Line 110 sets the value of X to an integral value randomly between 0 and 128. It also sets the value of Y to an integral value between 0 and 48.

Line 120 sets a point on the screen at the values generated in line 110.

Line 130 loops back to line 110.

TYPE: RUN <CR> and watch what happens. White dots will start appearing randomly on the screen as the program progresses. using the EDIT command from the last exercise, change line 100 to read

100 CLS: HIRES and RUN again. See the difference between HIGH and LOW resolution?

SCREEN ALLOCATIONS LORES MODE

```
SET 0,0        bottom left of screen
SET 0,47       top left of screen
SET 127,47     top right of screen
SET 127,0      bottom right of screen
SET 64,24      centre of screen
```

SAMPLE PROGRAM:

```
100 CLS
110 LORES
120 INPUT "TYPE IN COORDINATES X,Y ";X,Y
130 SET X, Y
140 GOTO 120
```

Line 120 causes the BASIC to request you to enter the coordinates of X and Y. You should the integral numbers you want for X and then for Y. The program will do the rest for you. Notice what happens if you enter coordinates greater than 127 for X and 48 for Y. Why not change line 110 to HIRES and repeat? Now the coordinates for X should not exceed 511 and for Y should not exceed 255. Look at the difference in resolution!

SCREEN ALLOCATIONS HIRES MODE

```
SET 0,0        bottom left of screen
SET 0,255      top left of screen
SET 511,255    top right of screen
SET 511,0      bottom right of screen
SET 256,128    center of screen
```

PLOTTING LINES

MICROWORLD BASIC contains a powerful command which uses the graphics facility to enable you to PLOT lines from one point to another. Try this problem:

```
100 CLS
110 HIRES
120 PLOT 0,0 to 511,255 to 511,0 to 0,0
200 GOTO 200
```

By now you should be able to grasp the BASIC programs with some measure of confidence. Line 100 clears the screen and line 110 selects the HIRES mode. The new expression is line 120 which is probably self-evident. You should check the coordinates above to see where the plot will run. Line 200 just creates a LOOP to stop the BASIC prompt (>) appearing on the screen.

TYPE: RUN <CR> and watch the results!

Try another variation on the theme! The command PLOT I will INVERT a PLOT that has been SET.

TYPE: GX/PLOT/PLOT I/ <CR>

TYPE: RUN <CR> and watch what happens.

A similar command is PLOT R which RESETS the condition on the screen.

3.7 MUSIC BY BEEthoven

The MICROBEE can easily be programmed to play musical tones under BASIC. The duration of each tone can also be set between 1/8 second and 255 times 1/8 seconds. The notes are as follows:

NUMBER	NOTE	FREQUENCY
0	REST	-----
1	A	220
2	A#	233
3	B	247
4	C	262
5	C#	277
6	D	294
7	D#	311
8	E	330
9	F	349
10	F#	370
11	G	392
12	G#	415
13	A	440
14	A#	466
15	B	494
16	C	523
17	C#	554
18	D	587
19	D#	622
20	E	659
21	F	698
22	F#	740
23	G	784
24	G#	831

Using this table it is easy to encode tones ar music directly. Try the following:

TYPE:

100 PLAY 10,40;20,80 <CR>

when RUN this program will play F# for 5 seconds followed by E

for 10 seconds.

Try the following program:

```
90  REM WHEN THE SAINTS GO MARCHING IN
100 PLAY 4,2; 8,2; 9,2; 11,10; 4,2; 8,2; 9,2; 11,10
110 PLAY 4,2; 8,2; 9,2; 11,4; 8,4; 4,2; 8,4; 6,10
120 PLAY 8,2; 8,2; 6,2; 4,8; 8,4; 11,4; 11,2; 9,10
130 PLAY 8,2; 9,2; 11,4; 8,4; 4,4; 6,4; 4,10
```

and type RUN <CR> to hear the tune. As you can hear, it is easy to make music on the MICROBEE!

3.8 STRING OPERATIONS

CONCEPTS

STRING ARRAYS
STRING VARIABLES
STRS
IMPLICIT STRING FUNCTIONS
    Simulating LEFT$
    Simulating RIGHT$
KEY$
CHR$

MICROWORLD BASIC has very comprehensive string handling capabilities which differ in some ways from other BASICS. STRING VARIABLES are simply REAL VARIABLES with a '$' sign. For example:

```
A0        REAL VARIABLE
A0$       STRING VARIABLE
B4(1,4)   REAL NUMERIC ARRAY
B4$(1,4)  STRING ARRAY
```

Note carefully that A0 and A0$ are NOT two - distinct variables and if you attampt to use both in a program, one will be 'lost'. In the case of an array, a given element may be either a real number or a string depending on the presence of the '$' sign. You do not DIMension string arrays in MICROWORLD BASIC. Instead, dimension a real array and use whatever elements you require as strings. Strings may be of any length up to 180 characters. String storage is dynamically allocated within an area of memory called 'string space'. After a NEW command the string space is set to 255 characters. If more is needed, use the STRS command to allocate more space.

STRS(2000) will allocate 2000 bytes in memory for strings.

We will now experiment with implicit string functions which is really a more complex way of saying working with portions of already defined strings.

This function takes the following forms:

```
str-var                                produces full string
str-var (int-expl, int-exp2, .....)    produces the full array
                                       string
str-var(;int-expA, int expB)             produces a portion of
```
the string (see below)

TYPE the following:

```
10 A1$ = "ABCDEF"
20 PRINT A1$(;3,6)
```

when RUN this will produce

```
CDEF   indicating that elements 3 to 6 were selected
```

Try the following:

```
10 A1$ = "ABCDEFGH"
20 PRINT A1$(;LEN(A1$)/2,LEN(A1$))
30 PRINT A1$(;LEN(A1$)/2)
```

will produce

```
DEFGH  selects elements from halfway to the end
DEFGH  selects the same as line 20
```

SIMULATING LEFT$

Several versions of BASIC incorporate a function LEFT$ which can be easily be simulated with:

```
A0$(;1,n)
```

Note: if you copied a program from another BASIC which incorporated LEFT$ functions you can use the GX command to correct the program as follows:

```
GX/LEFT$(A0$,4)/A0$ (;1,4/ <CR>
```

Its that easy!

SIMULATING RIGHT$

This function can be simulated with the following:

```
A0$(;LEN(A0$)-n+1,LEN(A0$))
```

Try this program

```
10 LET A0$ = "ABCDE"
20 PRINT A0$        produces .....ABCDE
30 PRINT A0$(;2)    produces ..... BCDE
40 PRINT A0$(;2,4)  produces ..... BCD
```

KEY$

MICROWORLD BASIC incorporates KEY$, a function which will read one character at a time from keyboard input. This is particularly useful with games and programs which only need a single keystroke without having to use the <CR> to enter.

Try the following:

```
10 A1$="    "            one or more spaces
20 M$=KEY$               defines A0$
30 IF A0$="" THEN 20     tests for nul string (i.e no input)
40 A1$=A1$+M$
50 PRINT A1$
60 GOTO 10
```

and see what happens. Note that line 40 concatenates the characters you have typed in and it is best to preset A1$ to a space or other character to avaoid problems with undefined strings. The test in line 30 is critical as it continues to loop back until a character is pressed. This sample program should serve as a basis for all your KEY$ routines.

CHR$

The CHR$(int) function returns a character which has the ASCII value specified by 'int'. It is the reverse of the ASC function.

Try

```
10 PRINT ASC(A)    produces 65
20 PRINT CHR$(65)  produces A
```

LEN (str-exp) , VAL (str-exp)

For completeness we will examine two string related functions.

```
10 PRINT LEN("ABCDEFG")   produces 7
20 PRINT VAL("6.80")      produces 6.80
30 PRINT VAL("Apples")    produces 0
```

Line 10 calculates the LENgth of the string ABCDEFG and you will recall that we have already used this function in an earlier

exercise.

Lines 20,30 illustrate the VAL function. Essentially VAL returns the value or converts the string to a "real" number. Line 30 makes the point that alpha strings have no numeric value.

3.9 ERROR TRAPPING

CONCEPTS

ON ERROR GOTO
ERROR L
ERROR C

Microworld BASIC now contains a powerful command which is only found in a few very advanced BASICS. As you will have no doubt discovered for yourself, MICROWORLD BASIC error handling means that when an error is discovered the program stops and the line number and eror type are displayed. This is particularly useful when debugging a program with errors which occur as soon as the RUN command is typed. However some errors occur during the execution of the program and it may not be very convenient to have the program stop and display the error message. One example of this is when the HIRES graphics are in use and you do not realise that all the PCG characters have been used up. Interruption of the program clears the screen (and all your graphics as well!) and displays 'Graphics error in line xx '

To avoid the clear screen, error message reporting routine the command 'ON ERROR GOTO line number ' has been included. Again we will use an example:

```
10 ON ERROR GOTO 1000
20 PRINT "Hello                missing end quote!
30 ......

1000 PRINT "ERROR": STOP
```

When run this program will print 'ERROR' and STOP as directed in line 1000. Now type CONT <CR> and watch what happens again. Next change the program to the following:

```
10 ON ERROR GOTO 1000
20 PRINT "Hello
30 ......

1000 PRINT "ERROR"
1010 FOR T=l to 200: NEXT T   (a time delay loop)
1020 GOTO 20
```

and RUN.

Notice that the program 'falls through' ,the ON ERROR GOTO statement in line 10 and displays 'ERROR'. After the time delay loop the program flow is directed to line 20 (ie AFTER the ON

ERROR GOTO statement in line 10) and the conditional error message routine is invoked again.

The point is that the command ON ERROR GOTO In is an automatic error trap which is reset every time an error occurs. In reality if 'In' is 0 then conventional error reporting occurs otherwise a jump to line number 'In' will take place when an error is detected.

What about identifying an error? That is indeed possible and MICROWORLD BASIC uses the following extensions:

```
ERROR C          for error code
ERROR L          to identify the line containing the error
```

try the above examples but change line 1000 to read:

```
1000 PRINT "ERROR IN LINE "; ERROR L
1010 PRINT "TYPE OF ERROR "; ERROR C
```

Actually we are now really performing the same routine as that used by BASIC on detecting an error. Still the exercise may be useful for identifying a special error and reacting accordingly. the ERROR CODES are listed below.

ERROR CODE	ERROR TYPE
1	LINE TOO LONG
2	UNPAIRED BRACKETS
3	MULTIPLE STATEMENT
4	OUT OF DATA
5	MISSING END QUOTE
6	FN NAME
7	VAR MISMATCH
8	NOTHING TO EXEC
9	ILLEGAL DIRECT
10	UNDER/OVERFLOW
11	KILL NON LINE
12	NON EXISTENT LINE NUMBER
13	MIXED MODE
14	PARAMETER SIZE
15	STACK OVERFLOW
16	GRAPHICS
17	OUTPUT OVERFLOW
18	OUT OF MEMORY
19	SYNTAX
20	ZONE
21	NEXT WITHOUT FOR
22	ILLEGAL VARIABLE
23	OUT OF STRING SPACE
24	UNKNOWN FUNCTION
25	ILLEGAL LINE
26	DIVIDE BY ZERO
27	INTEGER STRING
28	ILLEGAL RUN MODE

```
28              DIM SIZE
29              PROGRAM TOO LONG
30              BAD LOAD
31              ARGUMENT ERROR
32              GOSUB STACK
33              LINE NUMBER CLASH
34              CAN'T CONTINUE
35              OPTION NOT FITTED
```

3.10 FORMATTING PRINTING

Often it is necessary to format the output of a PRINT statement. MICROWORLD BASIC has built-in formatting for the following types of output:

```
INTEGER         [Iint int-exp]
REAL            [Fn1.n2 real-exp]
EXPONENTIAL     [Dn real-exp]
ASCII           [An int-exp]
```

Effectively these are equivalent to the PRINT USING moulds used in other forms of BASIC. Try some examples:

```
10 REM this program prints integer values
20 FOR A=l to 10
30 READ N
40 PRINT [I10,n]
50 NEXT A
1000 DATA 123,1,4567,12345,56,9,123,1,2,3
```

will produce

```
       123
         1
      4567
     12345
        56
         9
       123
         1
         2
         3
```

in other words [I10,D] has produced a 'MOULD' and printed all output in a field 10 characters wide. Note that if the field is not wide enough to contain all characters '**********' will be printed. Try this experiment with [I3,D] and find out for yourself.

```
10 REM this outputs real numders right justified
20 FOR A=1 to 10
30 READ D1
40 PRINT [F8.2 D1] this means print D1 in a field 8
```

characters wide and include 2 for a decimal point

```
50 NEXT A
1000 DATA 1,2.3,4.56,789.0,1.234,2.345,0,2.3,4,7
```

will produce

```
   1.00
   2.30
   4.56
 789.00
   1.23
   2.34
   0.00
   2.30
   4.00
   7.00
```

This is useful to format business programs.

similarly the [Dn real-exp] will print the output in exponential format in a file n+7 wide with n decimal places.

```
10 C0 = 123456.00/99
20 PRINT [D5 C0]
```

will produce

```
1.24703E+03
```

Lastly the ASCII format is useful in outputting a known character a number of times.

```
10 PRINT [A5 10]
20 PRINT [A60 42]
```

will produce 5 'line feeds' and then print '*' 60 times across the screen.

SUMMARY

These 10 exercises should give you some greater insight into the many ways you can use the power of MICROWORLD BASIC. Only with experience can you master the various techniques involved. Hopefully this tutorial has clarified issues that are almost impossible to discuss effectively within a technical manual.

Section 4.

Programming

SECTION 4: PROGRAMMING

4.1 ASSIGNING VARIABLES

We have discussed previously that VARIABLES are numbers or strings stored in memory that change in value during the execution of a program. To assign values or expressions to VARIABLES, the instruction LET is used. LET causes a variable to be replaced by the expression.

```
10 LET A1=B2
20 LET A1=6.543
30 LET A2$=B5$
40 LET X2$="HELLO THERE"
```

are all valid assignment statements. The LET statement is optional and if not present at the beginning of a line it is assumed to be 'implied'. Thus

```
10 A1=B2
20 A1=6.543
30 A2$=B5$
```

will each be interpreted by BASIC in exactly the same way as those in the previous list.

4.2 VARIABLE TYPES

We have already mentioned that MICROWORLD LEVEL II BASIC supports INTEGER and REAL NUMERIC VARIABLES and STRING VARIABLES. The type of variable is set by the NAME given to it. INTEGER VARIABLES can be defined by single characters such as A, T, X, Y, Z. REAL (floating point) VARIABLES are defined by a character (A-Z) plus a number (0-7) such as AI, Tl, G4, B7. STRING VARIABLES, on the other hand, must have the format: character (A-Z), number (0-7) , followed by $. Examples of STRING VARIABLES are A1$, B7$, X1$, V0$.

Only variables of the same type may be assigned to each other and you must avoid mixing REAL and INTEGER variables in the same expression. So

```
10 LET A1=B1$                INVALID
20 LET A1="HELLO THERE"      INVALID
30 A1$=6.543                 INVALID
40 X=B1+2450                 INVALID
```

are all INVALID because an attempt has been made to assign data which is not the same type as the variable.

MICROWORLD BASIC has a special function FLT which "makes" an integer into a floating or REAL expression. Conversely the

function INT will convert real expressions to integers.

Any attempt to mix 'integers' and 'reals' in an expression will result in a MIXED MODE ERROR. The question arises as to how the mode of the expression is actually set. The answer is that the BASIC determines the mode. Consider the following examples:

If an expression appears in an arithmetic assignment statement (LET), the mode is determined by the assigned variable.

```
LET I=3+J      INTEGER because I is integer
LET A0=3/16    REAL since A0 is REAL
```

In some cases the mode is determined by the context; that is the specific position of the expression within the statement structure sets the mode.

```
A1(I-J)        Array subscripts must be integer
SQR(2+3)       Argument of square root is real
```

There are specific cases where the mode can only be determined by pretesting the expression. To avoid this time consuming task a specific restriction is placed upon the program. If in a particular expression neither of the cases above apply, then the first character in the expression is checked. Should the first character be INTEGER then the mode for the entire expression will be set to INTEGER. Otherwise the REAL mode will be set. Expressions appearing in PRINT and IF statements fall into this category.

```
PRINT I+2                INTEGER expression
PRINT 2+1                MIXED MODE ERROR!
PRINT INT(3.2)           INTEGER expression
IF 6<I THEN GOTO 30      MIXED MODE ERROR!
IF I>6 THEN GOTO 30      INTEGER expression
```

4.3 INPUT/OUTPUT STATEMENTS

The next group of instructions we will discuss are those which are used to enter data into the program and those which output data from the program to the screen. AS a group these are called I/O INSTRUCTIONS.

READ and DATA

The fundamental input statement 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 Al$,C4,D6
```

and the form of the corresponding 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 and input 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 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 comma 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 variable A1$, read 1.04 and assign it to the NUMERIC VARIABLE C4 and read 345 and assign it to the numeric variable D6.

The DATA statement may be placed anywhere in the program, but your programs will run much more quickly if they are all placed together. 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, move to the next and so on. If a program instructs BASIC to read data when there is no more data to be read then an OUT OF 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 Al$="FRED JONES"
20 LET C4= 1. 134
30 LET D6=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 has been assigned. The following READ and DATA statements will cause a MIXED MODE ERROR because Bl is a numeric variable and "FRED" is a string.

```
10 READ B1
20 DATA "FRED"
```

Note that string data MUST be enclosed by double quotes.

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 A1$,C4
30 RESTORE
40 READ Z1$,F2,G6
```

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

```
Al$ will be "FRED"  C4 will be 1.04
21$ will be "FRED"  F2 will be 1.04  G6 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 STAR TREK indeed that required all the commands to be written as DATA statements and included in the program. The INPUT statement 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 assigned to the variables. The form of an INPUT statement is

```
20 INPUT "WHAT IS THE NUMBER "; G1
```

INPUTTING 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.

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

When the RUN command is given, all variables of the form A1, B7 are set up as reals with value 0., so all strings must be assigned to before they are inspected, or an illegal variable error will result; they are NOT initialised to null strings.

```
10 INPUT A1$
20 IF A1$="" THEN PRINT "A NULL STRING HAS BEEN ENTERED"
```

will print

```
A NULL STRING HAS BEEN ENTERED
```

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

INPUTTING COMMAS

BASIC interprets a comma as a delimiter between places of data, so that if a 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

```
JONES, FRED
```

in response to the statement

```
30 INPUT A1$
```

will result in Al$ containing "JONES" 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 data. Hence

entering "JONES, FRED" will cause Al$ to contain "JONES, FRED". Another way to allow entry of any character is to use the "KEY$" string function, and build the input string in the BASIC program.

INPUTTING MORE THAN ONE VARIABLE AT TIME

The INPUT statement is not restricted to inputting only one variable at a time. The line

```
50 INPUT A1$,B2,C3,D5
```

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.

The types of data in an INPUT statement may be mixed, that is both string and numeric (real and floating). However, the correct type of data must match the variable specified.

PRINTING MESSAGE IN AN INPUT STATEMENT

To ensure the unambiguous inputting of data, more prompting than a question mark is usually required. BASIC does allow printing of a message within an INPUT statement. The statement

```
20 INPUT "WHAT IS YOUR NAME ";N1$
```

will cause

```
WHAT IS YOUR NAME
```

to be printed on the terminal before the input of Nl$ takes place. If you entered the following program

```
10 PRMT(#)
20 INPUT "WHAT IS YOUR NAME "
```

the reply

```
     WHAT IS YOUR NAME #
```

will be produced. Hence we can alter the PROMPT after a literal with a PRMT command previous to the INPUT statement.

ABORTING FROM AN INPUT STATEMENT

A special case occurs when it is necessary to abort a program while it is waiting for input. If you press break (BREAK) or CONTROL C (^C) you will STOP the current program and reenter the BASIC in the PROGRAM ENTRY mode so that you can make changes, LIST etc.

```
     CONTROL C (^C)      STOP in order to re-edit program
```

PAUSING EXECUTION

If you press CONTROL S (^S) during normal program execution (including graphics programs), the current BASIC program is STOPPED temporarily and will resume when any other keyboard key is pressed.

```
     CONTROL S (^S)      PAUSE and RESUME program
```

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 terminal. The statement LPRINT will cause the output to go to the printer stream instead of the VDU terminal. (See OUT# for details on data flow for each stream).

An example of the PRINT statement is

```
     10 PRINT "HELLO";Al$,FN0(34.5)*C3
     20 LPRINT "CALCULATION COMPLETE"
```

where "HELLO" and "CALCULATION COMPLETE" are string constants, while Al$ is a string variable and FN0(34.5)*C3 is a user defined real function multiplied by a numeric variable.

The use of the semicolon between "HELLO" and Al$ means that these will be printed next to each other while the comma between Al$ and FN0 means that a TAB is inserted after Al$. Consider the following example.

```
     10 LET A1$="FRED"
     20 FN0=9.76*#
     30 C3=1.0
```

```
     40 PRINT "HELLO ";A1$,FN0(1)*C3
```

will produce the following on the VDU

```
     HELLO FRED     9.76
```

FORMATTING OUTPUT

Although the use of a comma to separate the items in a PRINT statement will result in the items being printed in the next TAB or PRINT ZONE each time, this simple formatting has two shortcomings.

1. It may be desirable to use columns spaced more or less than TAB stops apart, and

2. When numbers are printed they are justified on the left hand side into the print zones and for integers the digits of equal significance may not line up.

```
For example    342                342
               1                    1
               1000 rather than   1000
               15                   15
```

(See also the later section for formatting print statements).

TAB

The first problem in setting up print columns different to the print zones provided can be solved using the TAB function. Thus

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

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

The TAB operand in the brackets can be a NUMERIC VARIABLE or a NUMERIC EXPRESSION, however it must be an INTEGER. If for any reason you are working with REAL VARIABLES then use the INT function to produce the operand for the TAB. For example

```
     10 FOR A1=.5 to 9.9 STEP .3
     20 PRINT TAB(A1);"*"
     30 NEXT A1
```

will not work, and an error message will show for line 20. But a slight change will produce the desired result

```
     10 FOR A1=.5 to 9.9 STEP .3
```

```
20 PRINT TAB(INT(A1));"*"
30 NEXT A1
```

Note that the use of TAB(0) will advance to the 256th column.

ZONE

MICROWORLD BASIC, in addition to the TAB features also lets you vary the ZONE width in PRINT statements. This is especially useful in formatting tabulated data which may require a different presentation to the normal zones.

The COMMAND is ZONE(int). The value of the 'int' sets the ZONE WIDTH used when commas are used in PRINT statements and the value may be any integer from 1 to 16.

4.4 MATHEMATICAL OPERATORS

Five mathematical operators are supported by this BASIC. They are:

```
+ ADDITION               reals or integers
- SUBTRACTION            reals or integers
* MULTIPLICATION         reals or integers
/ DIVISION               reals or integers
^ EXPONENTIATION (raising to a power, reals only)
```

Addition and multiplication are straight forward operations on the 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 integer variable A having a value of 1. Division is performed similarly. The first expression is divided by the second. Thus

```
20 LET A1=200/20
```

will result in A1 having a value of 10.

Integer division is quite different from REAL division in that the answer will be truncated towards the nearest integer to zero. Thus

```
10 LET A=-7/4
```

will result in A being assigned the value -1.

Exponentiation is simply a means of raising a number to a power and is usable with REALS only. Thus

```
30 LET Al=2^3
```

will result in Al having a value of 8. Note that greater speed and accuracy can be obtained for x^n if n is a small positive integer by successive multiplying e.g. 4^3=4*4*4.

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

```
40 LET A1=B5-4*5/6+C0
```

will be resolved by BASIC provided B5 and C0 have previously been assigned a value. If values had not been specifically assigned to B5 and C0 by an assignment statement, then they will be at ZERO, the value to which all numeric variables are set when BASIC is initialised.

PRIORITY OF ARITHMETIC OPERATIONS

Expressions are evaluated from left to right with parenthesis used to alter the order of evaluation. Standard algebraic precedence is followed:

i.e. the order of operations is

1. parenthesis
2. ^, negation and arithmetic functions
3. * and /
4. + and -

This is best illustrated with examples:

```
10 PRINT 2*3+1          * is done first
produces 7
20 PRINT 2*(3+1)        + first (parenthesis)
produces 8
30 PRINT 2*4^1.6        ^ first
produces 18.3792
```

As you can see BASIC does all arithmetic the way you would do it normally. PARENTHESIS (brackets) can be used to group parts of an expression if it is necessary to force BASIC to evaluate the expression in a different order. BASIC will evaluate the expressions in the innermost parentheses first and successively work outward. If you are not sure which order BASIC will evaluate a particular expression 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 parenthesis will make the meaning of the expression much clearer.

4.5 STRING OPERATOR

MICROWORLD BASIC has the ability to enable you to "connect" two strings together. This is called CONCATENATION. The operator for the concatenation of two strings is the "+" sign. An example best illustrates the point:

```
10 LET A1$=" MILLS"
20 LET B2$="PETER"
30 LET C2$=B2$+A1$
40 PRINT C2$
```

will produce

```
PETER MILLS
```

Further information on string operations are detailed in section 3.11.

4.6 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' in BASIC can be formed by using the GOTO instruction which transfers the execution of the program from the current line to the line specified in the GOTO statement.

```
100 GOTO 50
```

will cause BASIC to continue executing the program at line number 50. The following program will INPUT a number from the keyboard, calculate its square and square root and print out the values. At line 30 the GOTO instruction will cause the program to leop back to line 10 and repeat the whole process again.

```
10 INPUT "NUMBER";A1
20 PRINT A1^2,SQR(A1)
30 GOTO 10
```

In this program the loop will continue forever! It will never terminate. You can stop it by pressing the Break key or ^C, however the preferable way to exit from a loop such as this is to include a conditional branch out from the loop. This is discussed in the next section.

Note GOTO must be typed as one word and the line number in the GOTO statement must actually exist or an error message will result.

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

```
10 IF A=B THEN 1000
```

is the same as

```
10 IF A=B THEN GOTO 1000
```

CONDITIONAL BRANCHING

with the IF...THEN statement we have our first conditional branch instruction. This could easily be used to get out of the endless loop above. To do this the program becomes:

```
10 INPUT "NUMBER";A1
15 IF A1=9999 THEN GOTO 100
20 PRINT A1^2, SQR(A1)
30 GOTO 10
100 PRINT "END"
110 END
```

Another useful conditional branch instruction is the ON...GOTO. This takes the form

```
ON expr GOTO line_number_1, line_number_2, line_number_3
```

This is a conditional branch that depends on the expr in the following way:

```
expr.=1   Branch to line-no1
expr.=2   Branch to line-no2
expr.=3   Branch to line-no3
```

this is best illustrated with an example:

```
5 INPUT A
10 ON A GOTO 125, 230, 400, 650
125 REM HERE IF A=1
....

230 REM HERE IF A=2
....
400 REM HERE IF A=3
....
650 REM HERE IF A=4
```

Note that this differs from the earlier versions of MicroWorld Basic in order to bring it in line with more common practice in other BASICs. In MicroWorld Basics up to 2.7, the first line number would be branched to if the integer expression was equal to zero.

To run such programs, add a "+1" after the integer expression as it would have been used in BASIC 2.7 .

4.7 CONDITIONAL STATEMENTS _AN_D_ RELATIONAL OPERATORS

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

```
IF<condition> THEN <line number> ELSE <line number>
IF<condition> THEN <statements> ELSE <statements>
```

The <condition> may involve "<A less than, ">" greater than, "=" equal to or a combination of any two of these. The statement/s to the right of the "THEN" are only executed if the relational test is true. Otherwise, either the next numbered line or statements to the right of "ELSE" are executed.

If an assignment statement immediately follows either a THEN or an ELSE, the keyword LET must not be omitted, or the BASIC will think that you are trying to give it a line number to branch to.

Note that <condition> may also be an integer expression not involving relational operators, in which case the test is considered to be true if the expression is equal to -1 and false if the expression is equal to zero. This allows the use of BOOLEAN VARIABLES, or variables which indicate only true or false. Such BOOLEAN VARIABLES must be integer variables only.

Examples for use of IF .. THEN .. ELSE:

```
10 IF I<6 THEN 60
20 PRINT "YES"
.  .  .  .  .  .  .  .  .
60 PRINT "NO"
```

If I is less than 6 branching to line 60 will occur. If I is equal to or greater than 6, the program continues at line 20.

```
105  IF A0+6 >=B0 THEN LET I=0 ELSE LET I=I+1
110 .  .  .  .
```

If the value of the expression A0+6 is greater than or equal to B0 the statement to the right of "THEN" is executed, so I is set equal to 0. Otherwise the statement proceeds to "ELSE" and I is set to 1+1. Note if the "THEN" is executed the "ELSE" is bypassed and program proceeds to line 110.

The <condition> section generally 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 also =>

The expressions compared in a conditional state˜ent may be either string EXPRESSIONS on 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.

Each of the following statements are valid

```
10 IF A1$ = B1$ THEN 300
20 IF B1 = 1.4 THEN 320
30 IF SIN(2+B9) = COS(A2-4) THEN PRINT "REALLY?"
```

but

```
40 IF B1 = A2$ THEN GOTO 230       INVALID
```

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

WHAT HAPPENS IF THE TEST FAILS

If the relation tested is NOT TRUE, the program will first look for an ELSE statement on the same line.

If such an ELSE is found, statements to the right of it are executed, (or a linenumber is branched to).

If an ELSE is not found on the same line, execution continues at the next numbered BASIC line.
Statements following the THEN are never executed if the test fails.

4.8 FOR...TO...NEXT LOOPS

As mentioned earlier, the ability of the computer to perform repetitive tasks many times is one of its most useful features. A program which 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 A1=0
20 PRINT "-";
30 LET A1=A1+l
40 IF A1<= 80 THEN GOTO 20
9999 END
```

will print '-', increment the variable A1, and then if A1 is less than or equal to 80, will loop to do the procedure again. This will continue until Al is greater than 80. In effect this program will print a line containing 80 hyphens.

Because this type of loop structure where the number of times the loop is to be executed is known in advance, is so commonly used in programs, BASIC contains a specific instruction to do the task. This is called a FOR...NEXT loop.

```
10 FOR A1=1 to 80
20 PRINT "-";
30 NEXT A1
9999 END
```

This produces exactly the same result as the first example, but is much simpler. 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 (if the variables are correctly named~. The part of the instruction which defines the size of the increment or decrement in STEP. For example

```
10 FOR I1=80 to 40 STEP -1.5
....
....
50 NEXT I1
```

will execute the loop beginning with I1 = 80 and then DECREMENTING the value of I1 by 1.5 each time until I1 IS LESS THAN OR EQUAL TO 40.

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 internal 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.

See the NEXT* statement description in section 4 for one possible solution to this problem. The best solution is not to use FOR..NEXT loops for things that they are not suited to doing.

4.9 SUBROUTINES

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

This smaller program within a larger one is called a SUBROUTINE.

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.

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.

4.10 GRAPHICS AND ATTRIBUTES

The commands provided in the MICROBEE for controlling graphics and the inverse/underline attributes all use an item of hardware known as the PCG or programmable character generator. Some understanding of the principles involved is important to realize the limitations of the display system.

For graphics, or characters with attributes, the MICROBEE must create its own character in a 8*16 dot spacing and then display it in one of the 1024 possible character positions.

To generate the inverse and underline attributes, the CPU has access to the character generator ROM from which it constructs either inverse or underlined characters, but to be able have the full inverted character set on hand, the PCG is filled to capacity with inverted characters.

Therefore, it is impossible to mix attributes with each other or with graphics; an inverted line and an underlined line could not appear at the same time. Both types of graphics are also exclusive of the other display modes, because the use of the PCG for graphics means there is no room for other new characters to be created.

Due to these restrictions, there are five "MODE" select commands which select the relevant item exclusively ..

```
INVERSE        ;all output after this will be inverted
UNDERLINE      ;all output after this will be underlined
HIRES          ;selects HIRES graphics mode
LORES          ;select LORES graphics mode
PCG            ;select USER defined characters
NORMAL         ;all output will be reverted to normal
```

Graphics:

There are two different types of dot graphics:

Lores graphics:

Lores graphics are made up of chunky graphics rectangles which are set up in the same way as graphics on the TRS-80, with a resolution of 128 horizontal by 48 vertical, thus x=0 (left) to 127 and y=0 (bottom) to 47.

Mixing of normal text and graphics is freely allowed. If a dot is set on a character, that character space is cleared and the dot is set, while characters can be freely written onto existing LORES graphics, and LORES graphics may even be scrolled (in which case the current pattern’s y ordinates will increase.

Hires graphics:

Hires graphics provide a much greater resolution of 512

horizontal by 256 vertical (x=0 at the left over to 511 and y=0 at the bottom up to 255), but it also has important limits which cannot be ignored.

Because HIRES graphics is so fine, it is impossible to store enough characters to specify any desired configuration as is done for LORES graphics. (otherwise 16k bytes of memory would be required to achieve this density). Therefore, a complex program is provided which takes care of deciding what new characters are needed, which characters are no longer needed, ˜nd keeping track of how many free characters are left. This results in a graphics system with very high resolution but reasonable memory usage.

The maximum number of PCG characters available is 128, and if this limit is exceeded, an error will occur. This error can be avoided through the use of a special integer function of no arguments "USED" which returns the number of PCG characters used in the HIRES mode so far. This function should be examined at critical points in a HIRES graphics program, and steps taken to avoid a pcg full situation (indicated by a GRAPHICS ERROR). Another method is to use the ON ERROR GOTO instruction to trap the overflow and try something smaller or simpler.

The programmer must also be more careful with the use of normal characters when the HIRES command has been given. The program must not scroll the screen, or attempt to set HIRES dots where characters have been written, although it is perfectly alright to print characters over where graphics have been, and this can be used to label graphs, name parts in diagrams and for other mixed text/graphics applications.

The restriction on the amount of complexity that the HIRES graphics can have can be eased in se˜eral ways:

1) Use a smaller section of the screen for the graphics, as a good resolution will still be obtained with many fewer PCG characters used.
2) Draw (PLOT) horizontal and vertical lines rather than angled lines whenever possible.
3) Make sure that the HIRES command is reissued when a particular diagram is finished with, this will re-set up all the software, and set the number of PCG characters used back to 1.

Once the relevant command has been given to set HIRES or LORES modes, all of the commands for setting dots etc. are the same except of course for the different maximum values for the x and y co-ordinates.

On the MICROBEE, dot graphics are organized on the normal first quadrant cartesian system which is more compatible with the way people generally think about graphics coordinates than the system used on some other computers.

Therefore x=0 at the left hand side of the screen and increases to the right, while y=0 at the bottom of the screen and increases upwards.

This system should be used for all new graphics works, but to provide upwards compatibility with MicroWorld Basic 2.7 and ease of conversion from other computers' programs, it is possible to specify an inverted Y-axis by appending an "H" to the basic

graphics keywords SET, RESET, INVERT (this precludes double suffixes with PLOT such as PLOTIH, see Section 4 - PLOT) .

example:
SETH A+2,Z*3 is equivalent to SET A+2,47-Z*3 if we are in LORES graphic5 mode.

Basic graphics keywords

Set:

SET x,y is the general form of the command to turn on one dot in the relevant graphics mode. Note that x and yare integer expressions, so if you have a real number, use the "INT" transfer function to make it an integer. e.g.
SET X+INT(R9*COS(T0)),Y+INT(Rl*SIN(T0)) can be used for drawing circles, where T9 is run from 9 to 2*PI in a FOR loop, and R0 and R1 are the scaled radii, and X,Y is the co-ordinate of the centre of the circle.

Reset:

RESET x,y is the same but resets (turns off) the dot.

Invert:

INVERT x,y is similar but if the dot is on, it is turned off; if the dot is off, it is turned on.

PLOT:

This command can join pairs of co-ordinates by a line, and either set, reset, or invert all the points deemed to be on that line.
The basic form of the command is:

```
PLOT x1,y1 TO x2,y2
```

but may be expanded to do point to point graphics, e.g.

```
PLOT x1,y1 TO x2,y2 TO x3,y3 TO x4,y4 ...
```

The addition of suffixes to the PLOT keyword allows resetting and inverting of lines:

```
PLOTR x1,y1 TO x2,y2     ;will clear all points on the line
PLOTI xl,yl TO x2,y2     ;will invert all points on the line
```

Note that although PLOTH x1,y1 to x2,y2 is supported, and has the effect of inverting the Y axis, the subtractions must be done by the program when it is desired to to invert or reset points on a line (so as to avoid double prefixes, which are not allowed). e.g.

```
PLOTI x1,255-y1 TO x2,255-y2. (assumed HIRES mode).
```

Point(x,y):

The point keyword gives an integer function which returns a value depending on whether the dot is set or not.

If the co-ordinates are out of range for the relevant graphics

mode, the value returned is -1.
If the specified point is set, POINT returns -1.
If the co-ordinates are in range, and the dot is not set, POINT returns 0.

The values 0 and -1 were chosen to correspond to the two boolean valup.s for an integer which can be used in an IF statement directly, so

```
10   IF POINT(X,Y) THEN LET A=-A:B=-B ELSE SET X,Y
```

will negate A and B if the dot is set, and if it was not set, this statement will set it.

Direct PCG graphics

Although the HIRES and LORES graphics modes were designed to cater for most graphics'needs, it is often best to access the PCG memory directly using the POKE command and manually create the graphics qharacters. The direct PCG method is a good choice when there is a lot of text to mix with the pictures, for example in elementary arithmetic programs a sum of 2+3=5 could be illustrated by drawing 2 cars, a plus symbol, then 3 cars. Another reason for using the PCG directly is that of speed; since whole characters are handled at one time, drawing a picture becomes as fast as printing text.

In total, 128 PCG characters are available for use by any program at one time. Each of these characters may fill one or MORE of the 1024 character positions available on the screen. Just as the screen is broken up into 16 lines of 64 characters each, each character is broken up into fine dots, 8 across by 16 down. These dots are the same size as the dots used in HIRES graphics mode.

Thus, the first step in creating a set of characters to make up a picture is to decide how many characters wide and high it will be. Of course the characters are not square, the best way to see the actual ratio is to type INVERSE and experiment to see how many characters are required. Given this shape, dissect each character space up into 16 rows and 8 columns and draw in the shape as a rough outline (on paper !). From this rough outline, you must "digitize" the shape by deciding which dots will be on, and which will be off (shade in the "on" dots).
As an example, a representation of a car could be
as follows using 3 characters across the screen

```
(5 blank rows)
     **********
    *          **
 ***             ****
*                    **
*                      *
*****    *****    ******
     ****     ****
(4 blank rows)
AAAAAAAABBBBBBBBCCCCCCCC
76543210765432107654321O
```

Armed with this picture, the "PCG data" must now be determined character by character and entered in the program as DATA statements. Using the car example, look at character A first (the leftmost one). For each row of 8 dots in each character, one data value (which will become one BYTE in the PCG memory) must be calculated. This is done by adding a power of two for each dot that is on. The power is obtained by starting with 2A0 at the rightmost dot in the row to 2^7 at the leftmost dot in the row. As an example, take the 8th row of character A in the car example, which is ...

```
-     *    *    *    -    -    -    -         * on, - off
128   64   32   16   8    4    2    1         powers of 2
```

Since the 64, 32 and 16 valued dots are on, the data value for this row of the A character becomes 64+32+16=112.

Doing this operation for every row in character A, the data statement becomes ...

```
1000 DATA 0,0,0,0,0,7,8,112,128,128,248,7,0,0,0,0
```

note that there are 16 values, one for each row in the character.

Now assuming that there is a DATA statement for each character to be defined, these values must be POKED into the PCG memory to complete the definition process. The address of the start of the PCG memory is at 63488 and each character takes up 16 bytes (one byte for each row), so the address to start poking into is given by

63488+char code*16 where char code is in the range 0-127. For example, if-for the car graphics we want the first character to correspond to the capital letter "A", we look up the ASCII code for character "A" in the appendix and find that it is 65. Thus we start POKING at address 63488+65*16, and by putting the data statements for car characters Band C after the A data, these characters will correspond to the ASCII characters Band C.

To use the defined characters, use the PCG command and simply print the character which corresponds to char_code, e.g. PCG:PRINT "ABC":NORMAL will print one car.

Note that the PCG character definitions are lost when any of the other graphics or attribute modes are selected.

A full program to implement the car example is given in Section 5 : Applications Programs.

4.11 DEBUGGING AIDS

MICROWORLD BASIC has built in features to simplify ’debugging’ and modifying a program. The first is EDIT which enables the programmer to EDIT program lines already in memory.

To EDIT a line number in the current program in memory, type EDIT line-number <cr>. The line-number specified will be printed on the VDU and the cursor will be placed at the left-hand end of the line.

The following keys have special meaning during an edit:

^S	(CONTROL S) moves the cursor to the right
^A	(CONTROL A) moves the cursor to the left
DELETE	Deletes the character under the cursor
RETURN	The line to be reentered into the program file.

Most other keys will cause the corresponding character to be entered into the line to the left of the cursor (for full control key listing see section 4).

If you have attempted to run a program and an error message has been generated, you can edit the line at which the error occurred by simply typing EDIT <CR>.

Global eXchange

The GX command makes repetitive program editing easy by allowing the programmer to search for strings and either change them or leave them as they were, e.g.
GX 100 /PRINT/LPRINT/ <cr>
will prompt with each line after line 100 containing 'PRINT' and change it to 'LPRINT' if the 'period' key is pressed. (See section 4 : GX for more details).

TRACE ON, TRACE OFF

To study the sequence of a program you can type TRACE ON and the line number actually executed by the program is listed on the VDU between [ ] brackets. TRACE OFF removes the facility.

4.12 STRING OPERATIONS IN MICROWORLD BASIC

MICROWORLD BASIC has full string handling capabilities although in some ways it differs from conventional BASICs. STRING VARIABLES have the form of REAL VARIABLES with an attached '$' sign. For example

A0	REAL VARIABLE
A0$	STRING VARIABLE
B4(1,4)	REAL NUMERIC ARRAY
B4$(1,4)	STRING ARRAY

It must be carefully noted that A0 and A0$ are NOT two distinct variables and if you attempt to use both in a program one wilf be' 'lost'. In the case of an array, a given element may be either a real number or a string depending on the presence of the '$' sign. You do not DIMension a string array, instead dimension a real array and use whatever elements you want as

strings.
     Strings may be any length up to 255 chatacters. string storage is dynamically allocated within an area of memory called "string space". After a NEW command, the string space is set to 255 characters. If more is needed use the STRS command described later.

     String operations can be used with many of the MICROWORLD BASIC commands. For example the LET command takes the form ...
     LET str-var=str-exp
The assignment statement may be used for creating or transferring strings. String expressions consist of literals (strings within quotation marks), string variables, and concatenations of these using the '+' operator. portions of the already defined strings can be selected by using the implicit string function.

Examples:

```
10 A3$="***rubbish"+" more "+" junk***"
20 PRINT A3$
will print
***rubbish more junk***


10 DIM A5(13)          note that $ is never used in a DIM
20 A5(1)=13.35         this element will be REAL
30 A5$(2)="yahoo"      this element will be a string
```
note that it is now illegal to reference A5(2) because it has been assigned as a STRING VARIABLE.

IMPLICIT STRING FUNCTION

     References to string variables may be in one of the following forms:

```
str-var             produces the full string
e.g. PRINT A1$

str-var(int-expl,int-exp2,..)
                    produces the full string (array variable)
e.g. PRINT A5$(2)         (assuming AS has been DIMensioned)
```

Various portions of non-array strings can be produced with the following constructions:

```
str-var(;int-expA)
str-var(;int-expA,int-expB)
```

produces a portion of the string determined as follows:

int-expA only - from the character of the string whose number is given by the value of the integer expression int-expA to the end of the string.

or

int-expA and int-expB - from the character whose number is given by the integer expression int-expA to the character whose number is given by the integer expression int-expB

Confused? Well there's nothing like examples to make a point...

```
l0 A1$="ABCDEF"
20 PRINT Al$(;3,5)
```

will produce

```
CDE                 selects the elements from 3 to 5
```

```
l0 A1$="ABCDEFGH"
20 PRINT A1$(;LEN(A1$)/2,LEN(Al$))
30 PRINT A1$(;LEN(A1$)/2)
```

will produce

```
DEFGH       selects elements from about halfway to the end
DEFGH       note that lines 20 and 30 are equivalent
```

```
10 LET A0$="ABCDE"
20 PRINT A0$        produces....ABCDE
30 PRINT A0$(;2)    produces....BCDE
40 PRINT A0$(;2,4)  produces....BCD
```

SIMULATING LEFT$(A0$,N)

Several BASICS incorporate a function LEFT$ which we can now easily simulate with:

```
A0$(;1,n)
```

SIMULATING RIGHT$(A0$,N)

This function can be simulated with the following:

```
A0$(;LEN(A0$)-n+l,LEN(A0$)) which can be simplified to
A0$(;LEN(A0$)-n+l)
```

SIMULATING MID$(A8$,n,m)

The MID$ function can be directly translated using the following construction, but it is often possible to simplify to resulting expression:

```
A0$(;n,n+m-l)
```

SUBSTRING RESTRICTIONS

An important point to note about using the implicit string function for taking substrings is that you should not use other string expressions as a part of the int-expA or int-expB parameters. For example,

```
100 Bl$=A0$(;LEN(A0$)!2)
```

is O.K. because the string expression inside is the same as that which you are taking the substring of.

```
110 B1$=A0$(;1,LEN("HELLO"))
```

should not be used, but should be replaced with the following sequence of instructions

```
110 Z=LEN("HELLO") :Bl$=A0$ (;l,Z)
```

STRING CAPABILITIES

Strings can usually be treated in similar ways to numeric variables and can be used with IF, INPUT, PRINT statements. String expressions may appear as GOSUB and VAR arguments (See Section 4 - VAR).

```
10 GOSUB ("HELLO",6) 100
20 END
100 VAR (A0$ ,B)
110 PRINT A0$,B
120 RETURN
```

when executed, the program will produce the following output

```
HELLO 6
```

String expressions and variables can also appear in READ and DATA statements. For example

```
10 N0$="WORLD"
20 READ A0$
30 PRINT A0$
40 DATA "HEL"+"LO "+N0$
50 END
```

This program outputs as follows

```
HELLO WORLD
```

String expression can be compared using the normal relational operators (<, <=, >, >=, =, <>). The ordering is based on the ASCII code sequence (see appendix) which means that

dictionary order prevails. For example

```
10 IF "FRED" < "JANE" THEN PRINT "this will always print"
```

Note that in string ordering, nothing always comes before something, so
"ABCD" < "ABCDE".

STRING RELATED FUNCTIONS

Let us now introduce three functions that have string arguments. These are ASC, LEN and VAL.

ASC(str-exp) An INTEGER function that returns the ASCII value of the string argument. For example

```
10 PRINT ASC("A")   produces 65 (look in the appendix and check!)
```

LEN(str-exp) An INTEGER function that gives the length of the string expression. For example

```
10 LET J = LEN("ABCD"): PRINT J
```

will produce 4

VAL(str-exp) A REAL valued function that converts a string into a corresponding number. Examples

```
10 PRINT VAL("6.8")      produces    6.8

10 PRINT VAL("VIC")      produces     0

20 J=INT(VAL(24.6))      puts J=24
```

OTHER STRING RELATED FUNCTIONS AND COMMANDS

FRE($) A REAL function which gives the amount of string space still available. Example (if straight after a NEW)

```
PRINT FRE($)        produces   256.
```

STRS(int-exp) A STATEMENT used to set the string space. After entering the command, the string space will be set to the value of the integer expression. e.g.
STRS(3000) reserves space for 3000 characters

STR(int-exp) or STR(real-exp)
This function converts an integer or real expression into a string. For example

```
10 A=15: B0=13.57
20 Zl$="APPLES ="+STR(A) + ": VALUE (EACH) =$"+STR(B0)
30 PRINT Zl$
```

will produce

```
APPLES = 15: VALUE (EACH) = $ 13.57
```

KEY$

This function is a powerful one which enables the operator to input information via the keyboard without having to use the <CR> key. Essentially it returns a 1 character string when a key is pressed and a null string otherwise. For example

```
10 CLS
20 A1$=KEY$
30 IF A1$="" THEN 20
40 IF A1$="W" THEN l00
50 GOTO 20
l00 PRINT "YOU HIT THE LETTER 'W'"
```

Note carefully the loop 3˜ to 2˜ testing for a null string as this will occur most of the time.

CHR$(int-exp)

This function produces an ASCII character specified by the code number (int-exp). It is an ideal method of producting ASCII characters from their code, particularly the non-print or control characters (see appendix for all codes).

```
10 PRINT CHR$(65)
```

will produce

```
A
```

when run

4.13 SPECIAL INSTRUCTIONS

MICROWORLD BASIC supports a number of rather special instructions. These refinements facilitate such things as clearing the screen, positioning the cursor and also gaining access to the computer memory directly.

The best way is to briefly discuss each instruction however it is a good idea if you check out the operation of each on your own computer.

CLS This instruction clears the screen. It can be used in the IMMEDIATE MODE or under a line number and Is used to "rub out" the entire screen, positioning the cursor at the top left hand

side of the screen, and then actually turning it off until the next print statement.

CURS The CURS command allows the positioning of the cursor (which controls where the next character is printed). There are two forms of CURS. CURS x,y where x=1..64 (across the screen) and y=1..16 (down the screen) for most applications, and CURS p, where p=0..1023 which is useful in running programs designed for other computers with "PRINT AT" statements.
example:
```
10   CURS 1,16
          puts the cursor at the left of the bottom line.
```

SPEED This useful instruction can be used to control the speed of BASIC writing to the vdu screen. It is especially useful prior to LISTing a long program because you can slow down the display to read the listing in more detail. The format is SPEED n where n is any number between 0 and 255. SPEED 0 is fastest and SPEED 255 is slowest. The "default" setting of speed after a COLD START is 0, or fastest.
example:
```
>SPEED 20 : LIST :SPEED 0<CR>
```
will list the’current program slowly, and then change back to normal speed when it has finished.

POKE int-exprl,int-expr2 writes a byte of data specified by int-exp2 into RAM memory specified by int-exprl. For example

POKE 100,255 will set RAM location 100 (in decimal) to 255 decimal (FF in hexadecimal).

Some useful locations to POKE

220	cursor control - poke 111 (normal), 1 (steady block) , 97 (flashing block), 15 (steady underline)
140	BREAK disable byte - poke 1 to disable the BREAK key, poke 0 to restore break action
162,163	RESET jump address - controls what happens after a RESET or power on - POKE 162,30:POKE 163,128 to automatically RUN the current BASIC program, POKE 162,33:POKE 163,128 to restore normal RESET action.
61440..	screen memory starts here
63488..	PCG memory starts here

Warning: be very careful POKEing round in memory because it is very easy to write over a critical scratch pad (the BASIC’s private memory) and cause erratic operation of the current BASIC program.

PEEK (int-expr) An integer "function", PEEK reads the data byte stored in the memory location specified by int-expr. For example

```
10 POKE 100,255
20 PRINT PEEK (100)
```

will cause the contents of location 100 to be printed on ths VDU 255

EXEC When a machine language program has been loaded using the LOAD command, this command is used to jump to the start address of that tape. Although an error message is given when no machine language program has ever been loaded, it is unwise to use EXEC unless you are sure that a program is still intact, and that its program space has not been corrupted by the BASIC.

4.16 INPUT AND OUTPUT REDIRECTION

MicroWorld BASIC has an extremely flexible method of controlling the flow of data to and from the keyboard, vdu, cassette port, RS232 port and parallel port. The idea involved is call "redirection" and simply means in the case of output that the data which would normally be going to the screen is siphoned off to some other device. In the case of input, it means that input characters that would normally have come from the keyboard can be accepted from other devices instead.

Redirection ensures that all the familiar commands (PRINT, INPUT, LIST etc) can be used to control such operations as saving and loading data with the cassette recorder, merging programs and controlling a printer.

Two commands control the selection of input and output devices, which are identified by an integer in the range 0..7.

For output, there can be any combination of devices selected at one time, including the case of no devices selected. The statement which controls device selection and deselection is

OUT# int-exp {ON}{OFF}

The form "OUT# int-exp" will select device number int-exp as the only output device.
The "OUT# int-exp ON" form selects device int-exp but leaves the status of all other devices unchanged.
The "OUT# int-exp OFF" form deselects device int-exp but leaves the status of all other devices unchanged.

The selection of input is similar, but selecting two devices at once will only work in the case of devices 0 and 1 (Scanned keyboard and parallel port input).

IN# int-exp {ON}{OFF}

The form "IN# int-exp" will select device number int-exp as the only input device.
The "IN# int-exp ON" form selects device int-exp but leaves the status of all other devices unchanged.
The "IN# int-exp OFF" form deselects device int-exp but leaves the status of all other devices unchanged.

Each device number 0..5 corresponds to one device (devices 6 and 7 are not used in ROM BASIC) .

DEVICE 0

Normal MicroBee scanned keyboard in and VDU output device. Device 0 is automatically selected for input and output when the MicroBee is RESET (or turned on) .

The VDU output driver responds to the following control codes:

HEX	DECIMAL	FUNCTION
07	7	beep
08	8	backspace (move backwards one character)
0A	10	linefeed (move down one line)
0C	12	home (and clear screen)
0D	13	carriage return (cursor to hard left)
0E	14	forward cursor
0F	15	up cursor

DEVICE 1

Parallel port device. This device allows the use of the parallel PIO port available on the back of the MicroBee through a DB15 connector. This port is normally set up as an input device, but is changed to output when the OUT#1 (or OUTL#1) command is given.

Note that after a RESET/power on with BASIC 5.00 OR 5.10, the OUTL#1 (or OUT#1) command must be reissued or the MicroBee will "hang" when an LPRINT is executed. Full details of the parallel port hardware interface and pin assignments is available in the technical literature.

DEVICE 2 Cassette at 300 baud.
DEVICE 3 Cassette at 1200 baud.

Devices 2 and 3 provide a means of saving characters onto a cassette recorder by buffering groups of characters together and sending them as checksummed blocks.

The block nature of the cassette dumping can often be ignored, because the BASIC interface with devices 2 and 3 is always on a byte-by-byte basis. In some situations though, extra time delays might be required to be inserted by the dumping program to allow sufficient processing time when the data is input. The blocks of data are terminated when either a CARRIAGE RETURN character is given or when the block length so far exceeds 255 bytes. The carriage return character is part of the sequence which creates a NEW LINE and is therefore given after a PRINT command has finished executing. The LINE FEED character is never transmitted to the tape recorder.

What this means to a BASIC program is that the output of one PRINT statement (or several if the semicolon is used to inhibit the NEW LINE) will be sent to the tape recorder as one block of data provided the line is shorter than 255 characters. If more than 255 characters are present on one line, then the line will be broken up into several blocks.

To provide adequate input processing time between blocks, a number of nulls (rubbish characters) proportional to the length of the block just sent are added after each block. In most cases (including that of MERGING programs) this time delay will be sufficient, but if loss of data is occurring, then extra FOR..NEXT delay loops can be added between PRINTing each line.

When device 2 or 3 is used as the input device, the MicroBee waits for blocks of checksummed characters from the tape recorder, but only accepts those which have no errors. Blocks with detected errors are discarded, the corrupt data is not "seen" by the BASIC (see later for suggested data format).

If there appears to be trouble with the first block on the tape, then it is probably due to an "automatic level control" in the tape recorder interfering with the recording level. To get over this problem, try sending some "dud" first lines before sending any data lines. Of course, if you do this then you must be prepared to throwaway the duds lines on input. For example:

```
on output ...
110 OUT #3
120 PRINT "!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!":REM dud line
130 PRINT "!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"
140 PRINT "*": REM start block
and so on dumping the real data

on input ...
110 IN #3 : OUT #0 : OUT #0 OFF
120 INPUT Z7$ : IF Z7$<>"*" THEN 120 : REM throwaway dud blocks
and so on .. read input data
```

If the control character with code 26 is sent to the tape device, then an "IN#0":OUT#0" will be automatically executed when that character is received later on. This feature is called "control Z release" and is used to turn the keyboard/ VDU combination back on after an automatic data transfer (e.g. MERGE).

Note that the room for the buffer used to "block" and "deblock" the tape data is borrowed from the HIRES graphics scratchpads, so the HIRES graphics mode is ALWAYS cleared when device 2 and 3 are used. (The memory used is from 300H to 3FFH, and to "force" HIRES mode back on if the above memory is saved in an array put a 4 into location E5H).

DEVICE 4 RS232 at 300 baud.
DEVICE 5 RS232 at 1200 baud.

Devices 4 and 5 control the standard RS232 port available at the DB25 connector on the back of all MicroBees. In both cases, the format used is 1 start bit, 8 data bits, 2 stop bits (no parity is generated or expected).

RS232 output is allowed whenever the CTS input line is at a high voltage level. If this line if low, then the MicroBee will wait until it goes high before starting to send a character. When

no connection is made to the CTS input, an internal pullup resistor allows RS232 output to proceed.

In BASIC 5.00, no input synchronization output is provided by the software. In BASIC 5.10 .., the line labelled as CLOCK and available on pin 24 of the DB25 (RS232) connector is used to indicate when the MicroBee is "looking for a character". This line goes to +5v (active) after the RS232 input routine is entered and the serial input line has gone to the MARK condition (i.e. when pin 3 of connector is at <= 0 volts). When a start bit is detected, the CLOCK line returns to 0v (non-active). Therefore, the CLOCK line may be used to indicate to the sending device when it is permissible to send. When connecting two MicroBees together via RS232, the CLOCK line output (pin 24) of one MicroBee can be connected to the CTS input (pin 5) of the other MicroBee (these lines will therefore cross like the data pins 2 and 3).

If this input synchronization is not possible then the sending device must be sure to allow enough delay for the MicroBee to complete its processing of the last line INPUT.

When the RS232 device (#4 or #5) is selected as an input device, the break function is not performed by the break key on the keyboard, but by the MicroBee looking to see if the line is "SPACING" (i.e. if RXD, pin 3 is at a voltage level >= 5 volts. While this is convenient when running the MicroBee remotely by a terminal, it can be a real nuisance in debugging programs which accept data from the RS232 line using device #4 or #5.
To disable the BREAK function use the following statement:

```
POKE 140,1
```

To re-enable the BREAK after IN#0 has been reselected, use:

```
POKE 140,0
```

practical examples of the use of I/O redirection

Merging programs:

Merging of two programs is possible using redirection to the cassette device. Firstly, set up the file to go at the end of the final program so that the linenumbers do not clash (use RENUM), and then save it at 300bd using the following line:

```
OUT#2:LIST:PRINT CHR$(26):OUT#0 <cr>
```

... then load in the file which goes at the beginning of the final program and RENUMber it to avoid conflicts with the second part. To then merge in the second part, type

```
IN#2
```

... and watch the lines being entered one by one as if you were typing them in (but without the finger-wear).

If lines are being missed, instead of typing "IN#2", type "IN#2:OUT#0:OUT#0 OFF" in which case time will be saved by not echoing the input lines onto the screen.

Normal keyboard input is restored when the CHR$(26) character is received back off the tape, but if the MicroBee appears to "hang" for some reason, just use the RESET button. If the tape recorder is good enough to handle 1200bd, then

substitute device 3 for device 2 in the IN#/OUT# commands (but ensure that no lines are being missed due to the shorter time available to enter the line).

Printer selection:

In addition to the OUT# command, there is another command called "OUTL#" which has the same syntax as OUT#, but operates on the printer output stream, i.e. that data which comes from LPRINT or LLIST commands. This means that various types of printers can be accommodated using the same LPRINT/LLIST commands. The default printer stream is device 5, i.e. RS232 at 1200 baud. If you have a serial printer, this is the best device to use. Set up the printer for 1200bd, 8 data bits, no parity and 2 stop bits.

For parallel printers, use device 1 and the interface circuit (available on request), so type OUTL#l before using the LLIST or LPRINT commands (this command must be re-issued after a RESET with the parallel port device) .

For testing programs which use the printer, the "OUTL#0" command can be used to direct LLIST/LPRINT output to the screen.

Of course, the normal output stream (PRINT/LIST) may redirect its output to the printer by using "OUT#5 ON" or "OUT#1 ON" depending upon whether you are using a serial or parallel printer. Use "OUT#0" to turn the printer device off.

Saving data to the tape and a recommended format to use:

An important requirement in any computer is the ability to save results for later use, even though this will be done a lot less often on a cassette based system than on a disk based system. MicroWorld Basic provides for this, again through input and output redirection. All that is required to be done is to PRINT the data in some format, and then INPUT the data later on in the same way.

Consider the following SIMPLE example which saves and retrieves an array of 100 integers, (DIM D(100)) :

```
saving ...
00100 OUT#3 : REM select 1200 baud cassette output only
00110 FOR I=1 TO 100 STEP 5 : REM send 20 blocks of data
00115  REM now print 5 per line, separated by commas
00120  PRINT D(I);",";D(I+1);",";D(I+2);",";D(I+3);",";D(I+4)
00130 NEXT I
00140 OUT#0 : REM restore VDU output

loading ...
01000 IN#3:0UT#0:0UT#0 OFF: REM select no o/p devices, 1200 bd in
01010 FOR I=1 TO 100 STEP 5
01020  INPUT D(I) ,D(I+1) ,D(I+2) ,D(I+3) ,D(I+4)
01030 NEXT I
01040 IN#0 : OUT#0 : REM restore KB in / VDU out
```

notice the line 1000 when loading. The "OUT#0:0UT#0 OFF" command ensures that the question marks normally produced by the INPUT statement are redirected to nowhere.

The above example will work well most of the time, but it lacks error checking and file naming facilities. The following block format is therefore suggested (each partition is ONE character) ...

```
DUD BLOCKS (send about 3 to allow ALC's time to operate)
|! |! |! |! |! |! |! |! |! |! |! |! |! |! |! |! |! |! |! |!

BLOCK 1 (header block)
|$ |$ |$ |$ |$ |c1|c2|c3|c4|c5|n1|n2|n3|n4|n5...
```

where $ is the actual dollar sign used to identify the header block, n1... hold the name of the file, c1..c5 hold the number of record blocks to follow.

```
BLOCK 2 ... (data blocks)
|r1|r2|r3|r4|r5| ..... data in this record .....
```

where r1..r5 is the number of this record (to check that no blocks have been missed) .

```
LAST BLOCK (end of file block)
|? |? |? |? |? |
```

where the ? are the actual question mark characters (used so that the MicroBee does not lock up if a block is missed - this block flags the error).

This format relies on the fact that if an error occurs when INPUTting data from device 2 or 3, the block in error is ignored. Since the blocks are numbered, it is immediately obvious if an error has been detected.

To send the DUD blocks is simple, just use

```
FOR I=1 TO 3:PRINT "!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!":NEXT I
```

The way of sending the header block would be something like this

```
PRINT "$$$$$"; [I5 C] ;N1$
```

where C has the number of records to send, Nl$ is the filename.

To send the DATA blocks, use the following

```
PRINT [I5 R]; ...data to send...
```

where R is the number of this block

To retrieve the total number of records and the filename from tape, use the following

```
00100 INPUT H1$ : REM get header as a string
00110 IF H1$(;1,5) <> "$$$$$" THEN 100 : REM wait for header
00120 C=INT(VAL(H1$(;6,10))) : REM get total # records
00130 N1$=H1$(;11) : REM get filename
```

To extract the data from each line, use

```
01000 INPUT D1$ : REM get string with data in it
```

```
01010 IF INT(VAL(D1$(;1,5)))=R THEN D1$=D1$(;5) ELSE GOTO PANIC
```
where R is the number that the current record SHOULD be.

Connecting two MicroBees by their RS232 interface:

It it hard enough to make computers talk to dumb peripherals. To make two computers talk to each other is almost impossible!

In the MicroBee with 5.10 BASIC, however, things are a little easier because the majority of the interfacing software and hardware has already been provided in the form of the RS232 interface.

The connections which should be made via an RS232 cable are:

```
PIN 2  MBEE a ---) PIN 3  MBEE b        these connect the data
PIN 2  MBEE b ---) PIN 3  MBEE a
PIN 24 MBEE a ---) PIN 5  MBEE b        these synchronize
PIN 24 MBEE b ---) PIN 5  MBEE a
PIN 7  MBEE a ---) PIN 7  MBEE b        the signal ground
```

Given these connections, a program may be transferred as follows:

```
destination MBEE    >IN #5
source MBEE         >OUT# 5: LIST: OUT#0
destination MBEE    > PRESS RESET KEY TO REGAIN CONTROL
```

Data transfer is very similar to the case of talking to the cassette recorder via devices #2 and #3, but some additional techniques can be used to advantage in some situations (when communicating without synchronization as in a MODEM link, these can become very neccessary).

The first thing to note is that if the RS232 input routine is entered while the RxD line is SPACING, then the input routine will wait for the MARK (normal) condition to be re-established before trying to receive a new character.

The other thing to note is that it is possible, using direct IN and OUT commands, to manhandle the serial lines and use them as REQUEST TO SEND and/or ACKNOWLEDGE lines. The use of such a facility is up to the advanced programmer, but the basic operations are as follows:

Setting the output line to SPACE (clear bit 5 of port 2)

```
Z=(IN(2) AND (255-32)) : OUT 2,Z
```

Setting the output line back to MARK (set b5 of port 2)

```
Z=(IN(2) OR (32)) : OUT 2,Z
```

Testing the input RxD line (b4,port 2) to see if it is SPACING

```
Z=(IN(2) AND (16)) : IF Z<>0 THEN it is spacing
```

Testing the CTS input (b3,port 2) to see if ready to send

```
Z=(IN(2) AND (8)) : IF Z <> 0 THEN we are allowed to send
```

4.15 ERROR MESSAGES IN MICROWORLD BASIC

As we have discussed earlier, one of the most useful features of MICROWORLD BASIC is the comprehensive error reporting system. In all 36 error messages are generated in the BASIC. Generally, after you have typed in a program, you will type RUN. At this point the BASIC 'tests' each line before it is executed to check for potential problems.

Once an error is detected, BASIC stops and displays an error message together with a listing of the faulty line with either an inverse or underlined character showing where the error was detected.

If the MICROBEE was in a graphics mode when the error occurred, the screen will be cleared first so that the "error position" highlight can take place with inverse characters.

Note, although these error messages are quite comprehensive, certain types of errors can 'fool' the BASIC to reporting incorrectly. You should always think carefully about each mistake and treat the error messages as a good guide.

It is possible to trap errors and stop them from aborting execution of the current program by using the ON ERROR GOTO command together with the ERRORL and ERRORC functions. (See Section 4: ON ERROR GOTO, ERRORL, ERRORC.)

ERROR MESSAGE LISTING

The numbers specified before each error are the code numbers which are returned by the "last error type" integer function, ERRORC.

1) LINE TOO LONG
Either the RENUMber or Global exchange (GX) command found that if the desired operation were to proceed, the line length would exceed 184 characters. The requested change is NOT performed.

2) UNPAIRED BRACKETS
Look for' (' without ')' and vice-versa, or else brackets were expected but not found (as in the STRS command).

3) MULTIPLE STATEMENT
Look for a ':' on a line where it should not be, or you have put two statements on the one line without a ':'. In particular, DATA statements must be on a line by themselves.

4) OUT OF DATA
A READ statement has asked for more DATA than there is.

5) MISSING END QUOTE
Pretty obvious this one.

6) FN NAME
Function number out of range. Only FN0 to FN7 are allowed.

7) VAR MISMATCH
The data being passed between GOSUB and VAR statements does not match in number or type.

8) NOTHING TO EXEC
You have no machine language program at the auto start address.

9) ILLEGAL DIRECT
The command/statement you are using is illegal in the DIRECT (IMMEDIATE) mode.

10) UNDER/OVERFLOW
i) The REAL number BASIC just calculated is too large (>9.999..E+62) or too small (<1.0000..E-64).
ii) an integer greater than 65535 was entered.

11) KILL NON LINE
The line you just tried to delete doesn't exist !

12) NONEXISTENT LINE NUMBER
A GOTO or similar command refers to a line number which does not exist.

13) MIXED MODE
The line being interpreted has a mixture of integer and real numbers in it. The method of fixing the problem is often to use either an INT or a FLT function to correct the mode.

14) PARAMETER SIZE
One of the parameters (arguments) supplied to a command or function is outside the allowed range.

15) STACK OVERFLOW
This can be caused by a FOR...NEXT loop exiting before completion (see Section 4: NEXT*) . Another cause could be too many nested FOR...NEXT loops or expressions that are too complicated. Note that this fault may not be reported immediately; the error may for example occur when the expression evaluator tries to use the stack, but it is too full.

16) GRAPHICS
One of the graphics commands has detected some sort of error such as co-ordinate out of range, or PCG full of HIRES graphics (See HIRES).

17) OUTPUT OVERFLOW
Usually occurs when too much is printed on one line, or when strings are concatenated, and the resultant string is longer than 255 characters.

18) OUT OF MEMORY
Yes when you use up all of the available space this message appears. Check DIM sizes on multidimensional arrays as this can

soak up lots of memory. For example DIM A1(50,50) reserves 13,000 bytes!

19) SYNTAX
Recheck the correct syntax for the line showing the error.

20) ZONE
The TAB ZONE specified is too large (>16) or negative.

21) NEXT WITHOUT FOR
This error occurs when the variable used in the NEXT statement does not agree with the variable specified in the matching FOR statement (or there was no previous FOR statement) .

22) ILLEGAL VARIABLE
Produced when an attempt has been made to access an array variable as a normal variable or vice-versa (note that all arrays MUST be dimensioned).
Also, if a REAL variable of the form A0,C7 etc is referenced as a string (A0$, C7$) or such a variable which has previously been assigned as a string is referenced as A0,C7 etc then this error will be given.

23) OUT OF STRING SPACE
Not enough memory reserved for strings. Try assigning more using the STRS(num) command.

24) UNKNOWN FUNCTION
The word highlighted could not be identified as a valid variable, constant or function for the current mode. This can be caused by using REAL functions in integer expressions or vice versa and is fixed by using the INT or FLT transfer functions.

25) ILLEGAL LINE
The numbers 0 and 65535 are not allowable as line numbers.

26) DIVIDE BY ZERO
Check earlier FOR...NEXT loops and variables because whatever happened in the offending line was a division by zero or some very small number.

27) INTEGER STRING
In MicroWorld BASIC, all strings names must use the form A1$, C7$ etc. String names of the form A$ are not allowed.

28) ILLEGAL RUN MODE
This statement cannot be used in RUN mode.

29) DIM SIZE
Attempting to produce an array in RAM which requires more than the available memory space. (Multidimensional real mode arrays need LOTS of memory.)

30) PROGRAM TOO LONG
Either the program you just tried to load into memory was

too big, or else the RENUMber command found that if the program were to be renumbered in the manner desired, it wouldn't fit in the available RAM space. (The program is then left totally unchanged).

31) BAD LOAD
The program which MICROBEE just tried to read off tape had a checksum error, so loading could not proceed. Try LOADing the tape again with a different volume or tone setting on the tape recorder.

32) ARGUMENT ERROR
Check the argument of the expression. Have you used a real where an integer is required? Or have you tried to find the square root of a negative number?

33) GOSUB STACK
Stack overflow due to GOSUB without a RETURN or too many nested GOSUB routines.

34) LINE NUMBER CLASH
The renumber program has determined that if the renumber was to go ahead, a line number would be produced corresponding to one already existing in memory. The program is left unchanged.

35) CAN'T CONTINUE
It is not possible to CONTinue after editing the program, clearing variables etc.

36) OPTION NOT FITTED
The desired command could not be performed because the required hardware was not connected to the MICROBEE.

Section 5.

Statement and Command Descriptions

SECTION 5: STATEMENT AND COMMAND DESCRIPTIONS

In MICROWORLD BASIC, statements and commands preceded by a number are entered into memory and become part of the current program. Line numbers may range from I to 65534. If a statement or command is entered without a line number, it is executed directly. Multiple statement lines are created by using the colon as a separator.

The following statements and commands are listed for reference in alphabetical order. Refer to the list detailed below for a definition of abbreviations used.

ABBREVIATIONS USED IN THIS SECTION:

char	ASCII character
int-var	integer variable
real-var	real variable
var	general variable
line-no	line number
int	integer number
rel-op	relational operator
str-exp	string expression
int-exp	integer expression
real-exp	real expression
exp	general expression
[]	ASCII space character
{ }	optional specification
<CR>	carriage return
^C	CONTROL KEY and C pressed together
^Q,^W,^A,^S,^Z etc	CONTROL KEY with the appropriate letter

AUTO {(} {int1 {,int2}} {)}

This command causes the BASIC to automatically prompt with line numbers as each line of code is entered, allowing entry of programs without having to type line numbers.

The user MAY provide one or two parameters after the word AUTO. If one parameter is given it will be the first line number inserted and the default step between successive line numbers will be used. If two parameters are given, the second will be used as the step between line numbers and this step size also becomes the default for later invocations of AUTO.

If no parameters are given, the program defaults to the line AFTER the last line entered with AUTO, and the default step size is used.

When the MicroBee is cold started, or the NEW command is given, AUTO defaults are set to a starting line of 100 with a

step of 10.
Brackets MAY be placed around the parameters if they are given.

To exit from the AUTO mode, simply type a null line <cr> only). If called again with no parameters, AUTO will restart at the number that it finished with the last time.
AUTO mode will automatically EDIT lines which already exist, so when an occupied line is reached, the EDIT mode is invoked. To exit this AUTO-editing feature, use the break key. (See EDIT command)

Examples of AUTO:

>AUTO <cr> (after a NEW) produces automatically inserted lines starting at 100 and incrementing in steps of 10 after each line of BASIC code is entered.

>AUTO 300,5 <cr> produces automatically inserted lines starting at 300 and incrementing in steps of 5 after each line of BASIC code is entered.

CLEAR

This command erases all values of all variables and strings and also erases any data structures such as array DIMensions, allowing arrays to be reDIMensioned. It can be used either in immediate mode or in a program.

Example:
```
00100 DIM A(200)
00110 CLEAR: DIM A(9)
```

CLS

This command clears the VDU screen and places the cursor at the top left hand corner.
CLS also has the effect of turning off the flashing underline at the cursor position, which means that graphics programs should use CLS before the relevant graphics mode keyword (HIRES or LORES) to keep the screen free of all but graphics.

Example:
```
00100 CLS : HIRES : REM cursor off, and prepare for graphics
```

CONT

This command continues execution from where it was stopped by a STOP command or by a BREAK, and in some cases when an error has occurred. As such it is very useful during program debugging.
The use of CONT is illegal, and the "Can't continue" error will be given if the program is modified in anyway after the BREAK or STOP is encountered.

CONT is allowed after an error has stopped execution, but since the arithmetic stack is reset by an error, the CONTinued program may not execute properly (often a NEXT without FOR error will occur).
See also TRACE and STOP.

Example:
```
>list
00100 INPUT A0 : B0=A0*2
00110 STOP
00120 GOTO 100
>run
? 24
Stop at 00120
>print b0
48.
>cont
?     ... and so on
```

CURS int-exp

CURS int-expl, int-exp2

The first form of this command places the cursor at the position specified by the integer expression. position 9 is at the top left hand corner of the screen, and numbering continues along the lines so that 63 is the top right hand corner of the screen, the second character line then beginning with 64. 1"’23 is at the bottom right hand corner. The expression cannot exceed 1923 without causing an error.

The second form of this command allows x-y cursor addressing, where int-expl specifies the column number from left (1) to right (64), and int-exp2 specifies the vdu line number from the top of the screen (1) to the bottom line (16).

In both forms of this command, the flashing underline at the cursor position is turned off after the cursor position has been moved.

Examples:
```
CURS 544 : PRINT "*";
```
prints an asterisk in the middle of the screen.

```
CURS 1,16 : INPUT "Press return for next frame"; A1$
```
puts a prompt on the bottom line and waits for a <CR>.

DATA exprl,expr2, ...

Provides data for a READ statement. Data must agree in mode (real, integer, or string) to the corresponding READ variable. Note that data values may be expressions as well as constants (including variable names). DATA statements must appear singly on

a line (you cannot put statements, including REM statements, before or after DATA statements) .
Note that string data MUST be enclosed by quotes.

Examples:

```
10 READ A,B0,A0$
20 DATA 10, 2*6.4, "Eat at JOES"
. . . .
```

after execution integer variable A will have the value 10, and real variable B0 will be 12.8, and string A0$ will be "Eat at JOES".

```
100 A0=3
110 READ B0
120 PRINT B0
999 DATA A0+A0
. . . .
```

after execution, A0 will have the value 3 and B0 will have the value 6.

DELETE I1 {,I2}

This command deletes ranges of multiple lines, for situations where large amounts of program are to be removed.
The first parameter, I1, is the line number of the first number of the block to be deleted. The second parameter, I2 is optional and specifies the last line number of the block of lines to be removed.
Any specified line numbers MUST exist to avoid accidental deletion of large blocks of program.
When used in a RUNning program, DELETE will delete the specified lines and then stop the program.

Examples:

DELETE 20,60 will delete lines 20 through 60 inclusive.

DELETE 40 will delete line 40 only.

DIM dim_list (see example)

DIM is used to set up storage for arrays of integers or real numbers/ strings. Arrays may have one or more dimensions, but the practical limit on the number of dimensions arises only in the amount of memory needed.
Note that in MICROWORLD BASIC, all array subscript (index) references begin at 0. programs whose subscripts start at 1 will work perfectly, but will waste one variable space.
The dimension arguments MUST be integer values but may be expressions.

The method used to set up string arrays in MicroWorld BASIC is slightly different to some other BASICS in that the dollar sign should not appear in the DIM statement. The individual elements are then assumed to be REAL numbers, but just as with normal variables, if a string assignment is made to an array element, that element becomes a string at that time. Therefore, it is often a good idea to include a FOR..NEXT loop which sets each of the array elements to a null string as in the second example.

Examples:

```
10 LET I=10
20 DIM A0(5) ,B1(10,10*I)
. . . .
```

for a string array

```
10 DIM S0(20)
20 FOR A=1 TO 20 : S0$(A)="" : NEXT A
. . . .
```

EDIT {line_number}

This command allows alteration of an already existing line number in the current user's program without retyping the whole line.

If no line number is specified, then the line edited is the last line which was either inserted, edited or in which an error had occurred. EDIT mode is also entered when the AUTO line-numbering routine detects a line which already exists (see AUTO).

EDIT mode is intentionally different from normal text entry because it is designed to make line correction easy, but within itself, it is "MODELESS", i.e. all keystrokes are independent and there are no "commands" or "insert modes".

When EDIT mode is entered, the specified line is printed onto the VDU and the cursor is placed at the left hand end of the line just after the line number. The user can then insert characters merely by typing them and some special control functions then control cursor movement and text deletion.

The following keys have special meaning during EDIT:

CONTROL S (^S) (or LINE FEED)	Moves the cursor to the right
CONTROL A (^A) (or BACK SPACE)	Moves the cursor to the left
DELETE	This key deletes the character under the cursor.
CONTROL L (^L)	Moves the cursor to the right, but changes upper case to lower case in the process.

CONTROL W (^W) Moves the cursor to the start of the next "word", which is defined as the first non-space character after the next space.

BREAK Abort the edit and leave the line unchanged (In case you change your mind) .

<cr> (CARRIAGE RETURN) Terminates the EDIT and the altered line is entered into the program file, just as if it were a new line. If you should alter the line number then the old line will remain as it was before the EDIT and a new line will be added.

any other key pressing any other key will cause that character to be entered into the line to the left of the cursor. Control (non-printable) characters are shown as underlines.

Example:

```
EDIT 100        (enter edit mode for line 100)
EDIT            (edit last line which was accessed)
```

END

The END statement is used to terminate program execution. No message is printed. Return is made to the command mode where the prompt ">" is given. It is illegal to CONTinue after an END statement has been executed, so the STOP statement must be used if it desired to CONTinue after the program halts.

Examples:

```
. . . .
00999 END

. . . .
00100 INPUT "Finished (yes/no)"; A1$
00110 IF A1$="yes" OR A1$="YES" THEN END
. . . .
```

EXEC

This command will jump to the "auto-start" address of the machine language tape which was last loaded into the machine. If no tape has been loaded since the last cold start, an appropriate error message is issued.

This command should only be used when the machine language program is still intact (BASIC can destroy machine language programs if the "top of memory" is not set by the machine

language program.)
(The EXEC address is stored in byte 166/7.)
Example:
```
>LOAD "TARGET"
>EXEC
```

FOR var=expl TO exp2 { STEP exp3}

FOR...TO statements are used to control looping. Their action is to bracket a group of statements which are to be repeated for a number of iterations which is known BEFORE the loop is entered. The NEXT* var construction provides a method of exiting such a loop before its "due time", but any way of exiting a "FOR..NEXT" loop is bound not to work in some BASIC, so such a technique is therefore not encouraged.

Example:
```
00010 FOR A=1 TO 10
00020 PRINT A
00040 CURS A*10
00040 NEXT A
```

In the above example, the statements on lines 20 and 30 are "bracketed" by the FOR on line 10 and the NEXT on line 40. Thus, the statements on lines 20 and 30 are repeated ten times, once for each value of the integer variable "A".

The syntax of the FOR statement is as follows. The FOR statement is followed by a variable name. This can be either a REAL (A0,B7 etc) or an INTEGER (A,Z etc), and the type of this variable determines the type that the "expl" start value, the "exp2" end value, and the "exp3" step value (if the variable must be increased in steps different to one) must be. Therefore, mixed mode is not permitted, and the INT or FLT functions would be needed if the types had be mixed.

After the variable name, an "=" character is specified followed by the start value, exp2. This specifies with what value the variable is assigned the FIRST time through the loop.
The "TO" separator is then followed by the termination value, exp3. This value is not necessarily the final value that the variable will take in the loop, but is rather the value which the variable will never exceed. This distinction has no significance when the STEP value is one, but consider the following example:

```
FOR A0=0 TO 1 STEP 0.3 ... NEXT A0
```

In this example, A0 will take values 0, 0.3, 0.6, 0.9, and the value 1 will not be given to the loop because 1.2 is greater than 1.

The STEP value is optional, and if it is not specified, the default value is 1. (REAL FOR...NEXT loops allow the use of fractional steps as in the above example.)

Integer indexed loops are substantially faster than real indexed loops.

Note that FOR..NEXT loops may be nested as far as stack limitations will allow (up to 24 for INTEGER loops, and 32 for REAL loops). In the following example, nesting goes only 2 levels deep (see how arrays and FOR..NEXT loops go hand in hand).

```
00010 REM Set up the array so all element have value 3
00020 DIM D(10,10)
00030 FOR I=1 TO 10 : REM OUTER loop
00040  FOR J=1 TO 10 : REM INNER loop
00050   D(I,J)=3
00060  NEXT J : REM If this was NEXT I, an error would occur
00070 NEXT I : REM This is the OUTER loop, so use I here
```

In this example, the use of "pretty-printing" is shown. This technique of adding an extra space shows the structure of a program, and the same concept may be used even in other types of loops which use the GOTO statement.

Some (in)compatibility problems to look out "FOR" ... The differences in the actual mechanism which various BASIC's use to control the FOR..NEXT loop mean that some incompatibilities exist.

The value that the loop control variable has during execution is important to some BASIC's. In MICROWORLD BASIC, alteration of the control variable will not cause any change in the number of iterations executed because MICROWORLD BASIC uses a "tripcount" which is calculated before the loop is entered and decremented each time through the loop (this increases speed by avoiding a comparison each time through the loop).

Thus, programs which set the control variable to its final value and GOTO the NEXT statement to exit a loop prematurely must be changed to use the NEXT* construction.

Finally, although it is generally agreed that the value of the control variable AFTER the loop has terminated is undefined, some badly written programs use the fact that in the BASIC for which the program was written, the value after the loop is the stepped total after it has been compared to the TERMINATION value and found to be greater.

In MicroWorld BASIC, the value after the loop is the stepped up value less than or equal to the TERMINATION value.

GOSUB {[exp1,exp2,...]} int-exp

Transfers execution to a "subroutine" with a label equal to the value of the integer expression, 'int-exp'. This subroutine is a group of statements which is terminated by a RETURN statement which makes the BASIC continue execution from the point after the original GOSUB. (See RETURN). GOSUB/RETURN is used to remove duplication of functions throughout a program (saving space), and also to modularise programs - breaking a

large task up into smaller sections.
     The optional expression list may be used in conjunction with a VAR statement to pass values to the subroutine. The number of expressions of arguments or arguments must not exceed the number of variables in the VAR statement. (See VAR below). Also, arguments must agree in mode to the VAR list. The effect is similar to performing a number of LETs before the GOSUB. Subroutine nesting is permitted. Exiting a subroutine with other than a RETURN will cause random data to remain on the stack, which takes up room on the stack as well as possibly causing "NEXT without FOR" errors.

Example:

```
     10 GOSUB 100
     20 PRINT "END"
     30 END
     100 PRINT "HERE IS THE  "
     110 RETURN
```

     This short program immediately transfers to the subroutine at 100 and prints the message "HERE IS THE ". On encountering the RETURN, execution transfers back to line 20, the statement following the original GOSUB. Here the word "END" is printed and the program terminates at line 30.

     See VAR examples for parameter passing with [] square brackets.

GOTO int-exp

     Transfers program execution to the line number given by the evaluation of the integer expression, int-exp. If the value of the int-exp is not a valid line number, an error condition results.

Example:

```
     10 PRINT "TESTING..."
     . . . .
     60 GOTO 10
```

When line 60 is executed, unconditional branching to line 10 occurs.

     Care should be exercised in creating such loops to provide some means of exit. Otherwise, you will have created what programmers refer to as an "infinite loop". Escape from such a loop can be accomplished by using BREAK, or CONTROL C (^C).

     Since the line number expression can contain variables, the GOTO may be used for conditional branching. The MICROWORLD BASIC GOTO instruction is thus referred to as a "computed GOTO".

     Another example shows a computed goto:

```
10 INPUT I
20 GOTO I*100

. . . .
100 PRINT "GAME 1"
. . . .

200 PRINT "GAME 2"
. . . .

300 PRINT "GAME 3"
. . . .
```

If in line 10, the user entered 2 for I, then the integer expression in line 20 would produce a value of 200. The program would proceed to line 200 and continue execution. other values of I would cause branching to different points in the program.

Computed GOTOs should be avoided wherever possible and the ON..GOTO construction used instead, because computed GOTOs cannot be renumbered, won't work with some other BASICs, and the code produced is generally non-robust (try entering 0 in above example). Convinced?

GX /search_string/replace string/

GX lnum /search_string/replace_string/

The GX command gives the user a global search and replace facility usually only available in large line editors. GX is an interactive command in that every time an instance of the string to be replaced is found, the BASIC waits for the user to enter either a period ..... to effect the replacement, or any other key to step to the next occurrence. Therefore, things such as variable name changes are easy to effect.

In the first form, the command is entered, the GX keyword is followed by a slash, then the string to search for in the case as it would appear in a LIST. This means that program keywords such as PRINT, INPUT etc. MUST be capitalised. After the search string, another slash is used to indicate the start of the replacement string. The replacement string is terminated by a slash. In the first form, the starting line is the first line of the program.

In the second form where "lnum" is specified, GX will start looking for the search_string at the beginning of line "lnum".

Given the line to start searching from, the BASIC will search for the first occurrence of the search string in the BASIC text. If such a string is found, the line in which the string occurred is listed with a highlight at the point of substitution, and the BASIC waits for a key to be pressed.

If the period key is pressed the change is effected.

otherwise, the BREAK key will abort the command, and any other key will cause the BASIC to skip the listed instance and then continue looking through the BASIC for another occurrence of search_string.

Example: change some PRINT statements to LPRINT statements.

```
>GX/PRINT/LPRINT/
```

Note that the "tilda" character "~" appearing in the search string will match any character in the program file. This allows-the alteration of expressions involving the "slash" division operator.

Furthermore, if a string of tilda characters extend past the end of an otherwise matching BASIC line, then a replacement is stilL possible and the matching string up to the end of the line is replaced. This makes possible some very clever manipulations.

Example: (who needs special programs ?)
DeREMming a program (keep an original with the REMark's please!):

For each of the following commands, hold down the period key to change all occurrences (or use it interactively).

```
GX/: REM~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~//
GX/:REM~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~//
GX/ REM~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~//
```

... where the string of tildas is as longer than the longest comment is likely to be. Note that some comments which are GOTOed will have to be replaced - try RENUMbering the program and reinserting REM lines till it RENUMbers successfully.

The inclusion of a tilda character in the replacement string causes a "slash" to be inserted at that point instead of the tilda, allowing replacement of strings involving the division operator.

The tilda and starting linenumber features are not available in BASIC 5.00.

HIRES

This command initialises the scratch RAM used for PCG HIRESolution graphics and sets up the screen so that no HIRES pixels (dots) are set.

HIRES must be used in the program before any SET, PLOT etc. commands are used. HIRES will wipe the screen, but will not affect the actual cursor position.

If HIRES is preceded by a CLS statement, the flashing cursor will be switched off before any graphics work is done.

Example:

```
00100 CLS : HIRES : REM set up to draw a Snoopy
... (add the rest as an exercise)
```

IF expr rel-op expr THEN statements { ELSE statements}
or more generally,

IF relational-expression THEN statements { ELSE statements}

IF...THEN is used to cause conditional execution of the statement or statements following the "THEN" or optional "ELSE". The relational operator, rel-op, may be < (less than), > (greater than), = (equal to) or a combination of any two of these, <= (less than or equal to), >= (greater than or equal to.)

The statement/s to the right of the "THEN" are executed only if the relational test is true. Otherwise, either the next numbered line or the statements to the right of the "ELSE" are executed. Examples make this clearer.

```
10 IF I<6 THEN 60
20 PRINT "YES"
. . . .
60 PRINT "NO"
. . . .
```

If I is less than 6, branching to line 60 will occur. If I is equal to or greater than 6, the program continues at line 20.

```
105 IF A0+6 >= B0 THEN LET I=0 ELSE LET I=I+1
110 . . . .
```

If the value of the expression A0+6 is greater than or equal to B0, the statement to the right of "THEN" is executed; that is I is set equal to 0. Otherwise, execution proceeds to the statement beyond the "ELSE"; I is set to I+1. Note that, if the "THEN" statement is executed, the "ELSE" clause is skipped and execution continues at line 110.

If an assignment is required after a THEN or ELSE, then the LET keyword must be used to distinguish the variable name from an implied computed goto.

In the more general case, the relational-expression can be made up from simple relations joined by "AND" and "OR" keywords and modified by the "NOT" keyword.

Relations negated by "NOT" which are part of a larger relational expression must be enclosed by brackets.

Examples:

```
00100 INPUT "Do you like IF statements ? " ; A1$
00110 IF A1$="YES" OR A1$="yes" THEN PRINT "That's good" ELSE
PRINT "I know when I'm not wanted !!!":NEW

00100 IF NOT (1=2 AND 2=3) THEN PRINT "peano rules"
```

Alternatively, an integer expression can be used as a relation. If the integer expression evaluates to -1, then the test is. taken to be true. If the expression evaluates to 0 then the test is taken to be false. This allows the use of BOOLEAN (or logical) variables.

Relational expressions with a value of 0 or -1 can be assigned to integer variables if the expression is enclosed by

brackets.

Example:

```
00100 A=(-1 OR  0) : REM true
00110 B=(-l AND 0) : REM false
00120 GOSUB 1000 : REM this one will print the message
00130 END
01000 IF A AND (NOT B) THEN PRINT "Condition satisfied"
```

IN# int-exp {on}{off}

This statement controls the source which the BASIC uses for all command line, INPUT and KEY$ entries. All input which would normally have come from the keyboard can be selected from 6 possible input streams. For a full description of the REDIRECTABLE I/O system and details of each device, see Section 3.14 .

The redirectable input scheme allows more than one input device to be selected at one time, but there is a restriction. Since devices such as the cassette recorder and the RS232 port cannot be "scanned" to see if a character is available, if one of these devices (2,3,4 or 5) is selected, then the device with the highest device number has the highest priority and all input will come from this source.

The input device to be selected is given as int-exp in the command precis, and specifies which device is to be either selected alone, turned on or turned off:

IN #x will select device x as the only device for the BASIC to take input from.
IN #x on will select device x and leave the status of all other devices unchanged.
IN #x off will deselect device x and leave the status of the other devices unchanged.

INPUT DEVICE NUMBERS:
0 Normal MicroBee keyboard input
1 External keyboard on PORT A of PIO (parallel port)
2 300 baud cassette
3 1200 baud cassette
4 300 baud RS232
5 1200 baud RS232

Pressing RESET will select device 0 (normal keyboard) as the sole input device.

Example: Dumping/ Loading array D(100) of integers simply.

```
 ...
00100 OUT #3 : REM 1200 baud cassette
00110 FOR I=1 TO 100 STEP 5
00120  PRINT D(I);",";D(I+1);",";D(I+2);",";D(I+3);",";D(I+4)
00130 NEXT I
```

```
00140 OUT #0 : REM restore vdu output
 ...
01000 IN #3: OUT #0: OUT #0 OFF: REM null out, 1200bd in
01010 FOR I=1 TO 100 STEP 5
01020  INPUT D(I),D(I+1),D(I+2),D(I+3),D(I+4)
01030 NEXT I
01040 IN #0 : OUT #0: REM restore kb in/ vdu out
```

INPUT { literal {;} {,} } var {,var}.....{;}

The INPUT statement is used to input data from the keyboard (or other input device) and to assign this to program variables. The optional literal can be used to print a message just before the inputting is to begin. The literal, if used MAY be followed by a comma, or a semicoion.

Without the literal option, the prompt character is always '?'. With the option, the prompt character is controlled by the user with the PRMT command. In the this case, the default prompt is a space. A semicolon placed after the last input variable will inhibit the <cr> and <If> that normally occurs after an INPUT statement.

Examples:

```
10 INPUT A, B, A0
20 PRINT A+6, A0*A0
30 END
```

The program immediately outputs a '?' and waits for the user to enter the appropriate data separated by commas.

If the user enters a <cr> before supplying all the required data, BASIC will print '??' indicating that more data is required.

If part of the user data is illegal (when inputting numeric variables), MICROWORLD BASIC will print 'R?' indicating that data should be reentered from the start of the input statement (i;e. start again with the data for variable "A" in the example).

```
10 INPUT "ENTER VALUE"; A;
20 PRINT A
30 END
```

In this program, the literal in line 10 is printed as a message followe9 by the space prompt (or whatever prompt character has been specified). The user then enters the value of A, and a <cr> to specify that he has finished.

Because of the presence of the ending semicolon, no <cr> and <If> will occur and the value of A will be printed beside the inputted value.

The inputting of strings is fairly straightforward except for the caSe where the user enters a comma as part of the data. The comma is used to separate the data to go to each variable in the input list, so if it is necessary for commas to be input, quotes can be used to surround the desired string, or else

(preferably), some arrangement using KEY$ could be set up to input in any format desired.

NOTE* MICROWORLD BASIC also distinguishes strings as integer and real. To avoid problems, the integer form (INPUT A$) has been disallowed and the 'REAL' form (INPUT Al$) must be used. An appropriate error message is generated at run time if an integer string variable is used.

INVERSE

When this command is executed, all VDU output will be changed to black on white format. If the last display mode was HIRES, LORES, or PCG, the screen will be cleared first. Note that it is impossible to "mix" INVERSE characters with UNDERLINEd characters or graphics because the PCG has been filled. Use NORMAL to revert back to white on black characters.

Example:
```
00100 PRINT "I like";:INVERSE:PRINT"inverse";:NORMAL:
PRINT"writing !!!"
```

INVERT{H}{[]} int-exp1, int-exp2

The INVERT statement toggles the state of a graphics point on the screen in either HIRES or LORES graphics mode. This means that if the dot was turned on, then it is turned off and vice-versa.

For LORES mode, int-exp1 must be in the range 0 to 127 and specifies a position across the screen, int-exp2 is in the range o to 47 and specifies a position UP the screen.
For HIRES mode int-expl must be 0 to 511 across and int-exp2 in the range 0 to 255 up the screen.

If the "H" option is specified, then the Y axis will be inverted. If the "H" option is not used, then a space MUST follow the keyword INVERT.

Example:
```
     INVERT 511,255
```
in HIRES mode will toggle the top rightmost dot on the screen.

LET var=expr

Statement used to assign the computed value of an expression to the variable to the left of the equal sign. The entire expression must agree in mode, either integer real qr string, with the assigned variable. LET is optional and can be deleted except for immediately after a THEN, or ELSE keyword. Examples:

```
10 LET I=2*6+3 After execution, I has the value of 15.
```

10 A0=U0+2.5 Assuming B0 has the value of 8.4 at the time of execution, A0 will be 10.9 after execution.

```
10 LET I=A0/2       ERROR...MIXED MODE
```

10 A0$="fred" After execution, variable A0$ will have "fred" as its contents.

LIST {l1} {, {l2} }

Lists on the VDU the current program in whole or in part, depending on the optional specification. Below are the 5 variations of the LIST command:

```
LIST         LISTS entire program
LIST l1      LISTS only line numbered l1
LIST l1,     LISTS from line l1 to end of program
LIST ,l1     LISTS from start to line l1
LIST l1,l2   LISTS from line l1 to l2 inclusive
```

Listing can be aborted at any time by striking the BREAK key (or ^C) and may be paused by CONTROL S (^S), hitting any other key will then continue the listing.

Example:

```
>LIST 100,190       Will list part of a program.
```

See also SPEED

LLIST

Same as LIST but output goes to the printer output stream which is controlled using the "OUTL #" command and can thus drive a printer if one is fitted.

The default printer output stream (set after a cold start) is 1200 baud RS232 out so as to suit a majority of serial printers.

An alternative to using LLIST/LPRINT commands is to select the printer device as "standard output" using the "OUT in command and then just using normal LIST/PRINT commands.

Example:

```
>OUTL #4: LLIST      Select 300bd printer then LIST program.
```

LOAD {U}{?} {"filename"}

Loads a file from cassette tape. If LOAD is used without a filename, then the BASIC will load the first BASIC or MACHINE LANGUAGE file found on the tape.

If a filename is specified, BASIC will look for the first file on the tape which matches the specified filename. The files

which do not match will be displayed, but will not be loaded.
     The filename can be up to six characters enclosed by " double quotes as for the SAVE command.

     If the "?" character is appended to the word LOAD, the BASIC will not load the program but merely verify that the checksums on the tape are O.K. (to verify a SAVE).

example:
>SAVE "MYPROG"
>LOAD?
will save the current BASIC program and then verify it.

     If the "U" or "u" character is appended to LOAD, the BASIC will ignore checksums when it tries to load the tape, allowing the user to recover programs where the error on the tape is small. This method will not always work, and if bytes are missed, some very strange things will happen when the program is listed and edited. (May need COLD start to recover.)
     This option should only be used as a desperation measure when you have been foolish enough not to dump two copies when SAVEing and then verify the first one.

BASIC files:
     MICROWORLD BASIC will load files with a 'B' file type (any file created with the BASIC SAVE command) into the BASIC program space, erasing the previous program at the same time, and prepare to edit or run the program just as if it had been typed in.

MACHINE LANGUAGE files:
     MICROWORLD BASIC accepts files of filetype 'M' as machine language files which are loaded into memory at the address specified on the tape, and optionally auto-executed (program starts automatically when loaded). Otherwise, the user may use EXEC to start up the program.

LOGICAL OPERATORS

     NOT, OR and AND can be used with relational operators in IF statements to perform logical operations with integer arithmetic. (See IF)
     The INTEGER representation of true is -1 and for false, it is 0.

Example:

```
     10 IF A<2 AND B>6 OR C<10 THEN 80
     ...
```

Note the operation precedence is
     (1) NOT
     (2) AND
     (3) OR
     parenthesis may be used to alter precedence.

The logical operation must appear with parenthesis when assigning to integers. Also, when sub-conditions are negated using NOT, that sub-condition should be bracketed as in (NOT Q) below.

Example:

```
10 I=(2<A AND 3>B AND (NOT Q))
20 IF I THEN PRINT "Condition satisfied if A=3,B=4,Q=0"
```

LORES

LORES will initialize the programmable character generator graphics ram and prepare to receive set, reset, plot etc. commands for low resolution graphics mode (128*48). Note that the screen is not cleared by this command, and inverse or underlined characters on the screen when this command is executed will change to random graphics characters. The screen is not cleared by this command, and if INVERSE or UNDERLINE characters were left on the screen, rubbish graphics will appear.

Once the LORES command has been issued, graphics characters may be produced by printing characters with codes greater than 128. The character corresponding to code 128 can not be printed because it is the internal code for "end of string", so a command PRINT CHR$(128) will not print anything. In LORES mode, this character would have been equivalent to a space, so substitute 32 (space) or 192 (graphics blank) for 128.

Example:

```
00100 LORES
00110 FOR C=129 TO 192: REM run over all graphics codes
00120 PRINT CHR$(C);: REM print char with code c
00130 NEXT C
```

LPRINT

Same as PRINT but output goes to the printer output stream which can be assigned to one of 6 actual output devices including RS232 (300bd/1200bd) for connection to serial printers, normal vdu output or even cassette out (see OUTL# and PRINT).

Example:

```
00100 OUTL#0 : REM when testing, print output goes to screen
00110 LPRINT "Calendar for year ";INT(Y0)
```

NEW

NEW erases the current program and all variables, resetting AUTO defaults to a start of 100 with a step of 10, resetting the number of REAL significant digits to 8 and other "housekeeping" tasks.

Example:
>NEW
>AUTO ...

NEXT var and NEXT*var line no

The NEXT var statement, companion to the FOR statement, tells the BASIC where the end of a group of statements to be repeated is located. (See FOR)

Note that the loop will NOT be terminated by setting the control variable to its final value and jumping to the NEXT statement as in some other BASICs. The NEXT* statement must be used to exit FOR...NEXT loops early.

GOTOing out of a FOR...NEXT loop can leave random data on the stack. The NEXT* command substituted at the point of exit will remove the unwanted values from the stack. Above, var is the index variable of the FOR...NEXT loop and line-no is the line number where execution should continue.
Example:

```
5  INPUT B
10 K=0
20 FOR I=1 TO 250
30  B>1 THEN NEXT* I 80
40  K=K+1
50 NEXT I
60 GOTO 5
80 PRINT "SUM=";K
90 GOTO 5
```

NORMAL

The NORMAL command clears INVERSE, UNDERLINE and PCG modes and thus returns PRINT output to normal format (white on black, non-underlined.)
Example:

```
00100 UNDERLINE
00110 PRINT "This will be underlined"
00120 NORMAL
```

ON ERROR GOTO int-exp

The ON ERROR GOTO statement provides a way of trapping errors that would normally cause execution to be aborted and an error message to be printed.

When int-exp is a line number, the statement will make the BASIC "remember" that when an error occurs, it is toM-7 try and GOTO the specified line number. If the line number does not exist when the error occurs, the program DOES abort with an error message. If int-exp evaluates to zero, then normal error messages are

re-enabled and any previous ON ERROR GOTO's are cancelled.

When an error does occur and is trapped to some line number, then the ON ERROR GOTO status is automatically cancelled and must be reset to provide further protection from errors. Also, the arithmetic stack is reset by an error, so execution CANNOT be made to resume from within a subroutine or a FOR loop.

See ERRORL and ERRORC under the INTEGER functions section for information on how to find out where and of what type the error was.

Example: Random lines with "out of chars" detection

```
10   ON ERROR GOTO 10
20   CLS : HIRES
30   PLOT 0,0 TO INT(RND*512) ,INT(RND*255)
40   GOTO 30
```

ON...GOSUB

```
ON int_exp GOSUB {[expr,...]} l1 {, {[expr,...]} l2 ... }
```

The ON...GOSUB construction is like a simple GOSUB statement, except that the subroutine called depends on the value of the integer expression int-exp and the list of line numbers. The line number to be used is the int-exp'th in the list, e.g. if int-exp equals 4, then the fourth line number in the list gives the address of the subroutine which will be called.

If int-exp does not correspond to a line number, no subroutine is called.

Example: Menu handling ...

```
00100 PRINT "Enter desired function
00110 PRINT "l:Enter invoice    2:List bad debts"
00120 PRINT "3:Fold             4:Spindle"
00130 PRINT "5:Mutilate"
00140 INPUT "-> ";J
00150 ON J GOSUB 1000,2000,3000,4000,5000
00160 GOTO 100
...
```

As in a normal GOSUB, the line numbers may be preceded by parameters to be passed to a subroutine using the VAR statement at the beginning of the subroutine.
Example:

```
00100 FOR I=1 TO 3
00110 ON I GOSUB ["boy","twistie"] 200,["girl","bread"] 200,
["MicroBee","Apple"] 200
00120 NEXT I
00130 END
00200 VAR (A0$,A1$)
00210 PRINT "The ";A0$;" ate the ";A1$
```

```
00220 RETURN
```

ON...GOTO

```
  ON int_exp GOTO line-no1, line-no2, {line-no3},
```

The ON...GOTO construction provides a conditional branch that depends on int_exp in the following way:

```
int-exp=1   Branch to line-no1
int-exp=2   Branch to line-no2
int-exp=3   Branch to line-no3     etc
```

The ON-GOTO differs from the computed GOTO in that the line numbers line-no1, line-no2, etc do not have to be calculated from the value of the int-exp.

If the int-exp is zero or greater than the number of line numbers given, then execution will continue with the next statement.

Note that this form is different from that used by earlier MICROWORLD BASICs which branch to the first line number when int-exp=0.

EXAMPLE 1: With computed GOTO

```
5 INPUT A
10 GOTO A*100 : REM this cannot be RENUMbered


100 REM HERE IF A=1
...
200 REM HERE IF A=2

1000 REM HERE IF A=10
.....
```

EXAMPLE 2: With ON-GOTO

```
5 INPUT A
10 ON A GOTO 125,230,400,650 : REM this will RENUMber
20 REM here if A<1 OR A>4
....
125 REM HERE IF A=1
...
230 REM HERE IF A=2
...
400 REM HERE IF A=3
...
650 REM HERE IF A=4
```

OUT int1,int2

Outputs the value of the integer expression, int2 as a data byte to a PORT with the address given by integer expression, int1. Of course, int1 and int2 must have values between 0 and 255 decimal. The most use for this command would be found when additions are made to the MicroBee which are not already supported by the BASIC software.

Example:
```
00100 OUT 0,1 : REM turn on the garden sprinklers
(hopefully connected to bit zero of the MicroBee parallel I/O
port after doing OUT#1 to set port to output.)
```

OUT# int-exp {on}{off}
and
OUTL# int-exp {on}{off}

The first form of this output redirection command controls the destination of all normal output which would normally go to the VDU (including all PRINT output, LIST output, error messages, prompts but NOT output from LLIST and LPRINT commands).

The second form controls the output which comes from LLIST and LPRINT commands, and the following comments apply equally to this form, except that "OUTL #" is used instead of "OUT #".

The normal output device and the LPRINT/LLIST output streams are totally independent.

The relevant output can go to none, some, or all of the 6 output devices (vdu output, parallel output, RS232 output at 300/1200 Bd and cassette output at 300/1200Bd.) For more details on the individual devices see Section 3.14 .

There are three forms of the command:

OUT{L} #x	this selects device x as the only device to receive output.
OUT{L} #x on	this command selects device x to receive output, but doesn't affect the state of the other devices.
OUT{L} #x off	this deselects device x leaving all other devices alone.

Null output can be obtained by deselecting all devices using an "OUT{L} #0:0UT{L} #0 OFF" sequence. When this is done, output goes nowhere which can sometimes be useful when testing programs or when INPUTting data from the cassette recorder using IN#.

OUTPUT DEVICE NUMBERS:

0 Vdu output device (normal)
1 MicroBee Parallel port output
2 300 Bd cassette output
3 1200 Bd cassette output
4 RS232 at 300 bd
5 RS232 at 1200 bd

For the normal (OUT #) stream, a RESET will select device 0 (vdu) as the only output device and device 0 (normal keyboard) as the only input device.

The default for the LPRINT/LLIST stream , set only at a COLD start, is 1200bd RS232 output so as to suit most serial printers.

Example 1: Output a file to tape for merging with AZ release
```
)OUT #3:LIST:PRINT CHR$(26) :OUT #0
 ...
```
Then when another program is in memory, use
```
)IN #3
```
to merge in the previously recorded file.

Example 2: Set up LLIST stream for a 300bd serial printer.
```
)OUTL #4
)LLIST .,.
```

PCG

All print output after this will use the PROGRAMMABLE CHARACTER GENERATOR. Note that the characters generated will have the PCG select bit set, so the PCG must be set up first. (See GRAPHICS in Section 3.10 and example program).

Example:
```
00100 LORES : REM set up PCG with lores graphics chars
00110 PCG:PRINT "This will come out garbled":NORMAL
```

PLAY n1 {,m1} {; n2 {,m2} ...}

This command will sound one or more notes from the speaker for multiples of 1/8 second.

The note "n" is the first number specified, and selects one tone from the 25 possible (see table). If 0 is used as the tone number, a tone is not generated, but a rest of equivalent time is allowed instead.

Example:
```
00100 PLAY 5 : REM sound C# for 1/8th second
```

The second number, "m" is optional and specifies the number of multiples of the basic time unit, 1/8th second to play the note for.

Example:
```
00100 PLAY 1,8 : REM sound A for 1 second
```

To avoid repetition of the PLAY keyword, it is possible to play more than one note by using the semicolon ";" to separate the note/length units.

Example:
```
00100 PLAY 2,3; 3,2; 0; 2,2; 3; 5,2 : REM rubbish tune
```

The 25 notes provided are the musical notes taken from the section of a piano starting at the A below middle C and then up for two octaves. The frequencies are reasonably precise and allow simple monophonic melodies to be played:

Number	Note	Frequency
0	rest	
1	A	220
2	A#	233
3	B	247
4	C	262
5	C#	277
6	D	294
7	D#	311
8	E	330
9	F	349
10	F#	370
11	G	392
12	G#	415
13	A	440
14	A#	466
15	B	494
16	C	523
17	C#	554
18	D	587
19	D#	622
20	E	659
21	F	698
22	F#	740
23	G	784
24	G#	831

```
PLOT{I}{R}{H}{[]} x1,y1 TO x2,y2 { TO x3,y3 {.... TO xn,yn} }
```

The PLOT command allows graphics commands to be performed automatically along straight lines joining two points. plot can be used in either LORES or HIRES graphics mode.

Plot will normally be used in the set dot mode which is the default for a PLOT keyword without suffixes.

Lines can also be inverted or reset by the addition of the letters "I" and "R" respectively after the PLOT keyword.

The usage PLOTH will plot a line with the Y-axis inverted, but note that it is impossible to have "PLOTIH" for example.

Note that if PLOT is used without any suffixes, then a space MUST follow the keyword or an error will occur.

If more than two x,y co-ordinates are specified as in the command precis, the desired graphics operation is performed on a point-to-point basis, making one continuous zig-zagging line.

Examples:

```
00100 HIRES
00110 PLOT 10,10 TO 100,100
00120 PLOT 0,0 TO INT(RND*512) ,INT(RND*256) TO 511,0 TO 0,0
00130 PLOTI 0,0 TO 511,255 : REM invert this line
```

POKE int-exp1,int-exp2

This command writes a byte of data defined by int-exp2 into RAM memory at the address specified by int-expl. Note that both integer expressions are of course in decimal. Some useful locations to poke, and the results are given in the appendix at the end of this manual and in Section 3.13.

Example: Change the cursor (use 111 to change it back).

```
>POKE 220,1
```

PRINT list

Statements used to output information and data to the console device (or other selected output device). The "list" consists of variables, constants, expressions, quoted literal and special printing functions separated optionally by commas, semi-colons or back slashes(\). A comma produces zone spacing with the size of the ZONE determined by the ZONE command.

If a PRINT is terminated by a semicolon, the final CR and LF <cr>,<lf> is suppressed. Backward slashes, or sloshes (\) may be used within the "list" to produce additional CRs and LFs.

The special print functions are TAB(int-exp) and SPC(int-exp). TAB moves the cursor to the position equal to the value of the integer expression. SPC produces the number of spaces determined by the value of the integer expression. These two print functions will produce the same result only when the cursor is "hard left".

Numeric values may be output in one of two ways: formatted and unformatted. Consider the latter first. Unformatted printing is that used in standard BASIC. If the value is between 0.01 and 999999, it will be printed in ordinary decimal notation. For smaller values and larger than these, exponential notation is used. Best we resort to examples:

```
PRINT 65+3              Produces 68.

PRINT 500000*2          Produces 1.0 E+6
```

```
10 PRINT I,J;A0*2
20 PRINT "OK"; TAB(8);
30 PRINT "DONE"
40 END
```

This program would produce the following output for the values, I=2, J=6, A0=12.4 and ZONE set at 14

```
2         6              25.8
OK   DONE
```

The formatted printing of numerical values is useful in business programming where specific fields must be set up to accommodate the printed values. The format specification may appear at any point in the PRINT "list". It takes one of four forms:

[Iint int-exp] Integer format:

The value of the integer expression is printed RIGHT justified in a field of width 'int'. 'int' must of course be greater than the number of digits in the value to be printed plus one (to take care of the sign). If this is not the case, plus signs will be printed in the field.

Example:

```
00100 A=123
00110 PRINT [15 A]
```

will print ' 123', i.e. two spaces plus three digits makes up the fieldwidth of five.

[Fn1.n2 real-exp] Real format:

The value of the real expression is printed in a field n1 wide and a decimal point n2 digits from the right of the field. The field width must be TWO characters greater than the number of digits printed. The number of digits printed should not exceed the number of significant digits. Again, plus signs will be printed in the field if the specification is incorrect.

Example:

```
00100 A0=45.23
00110 PRINT [F8.4 A0]
```

will print ' 45.2300', i.e. one space, two digits, a decimal point and four more digits making up a fieldwidth of eight. Note that the trailing 0's are printed because four digits after the decimal point were specified.

[Dn real-exp] Exponential format:

The value of the real expression is printed with exponential format in a field n+7 wide with n digits after the decimal point.

Example:

```
00100 PRINT [D4 45.23]
```

will print ' 4.5230E+01'. Note that the extra 7 characters which make up the fieldwidth are constant for this fieldwidth.

[An int-exp] ASCII format:

Equivalent to the PRINT CHR$(int-exp)

The value of the integer expression is output as its equivalent ASCII character a total of n times. This format is useful for sending special control characters to the output device, since the CHR$ function cannot for example print the character with code 128 since that is the "end of string" code. It also permits printing long strings of the same character.

Example:

```
>PRINT [A120 7]
```
     will ring the bell 120 times (the code of ^G is 7).

The format specifications may of course be linked with semicolons as below:

```
     10 PRINT [A6 66]; [F8.2 A0]
     20 END
```

For A0=42.6, this program would produce the output
BBBBBB   42.60

(See also LPRINT)

PRMT (char)

     If a literal appears at the beginning of an input statement, the prompt character following the literal is determined by the character 'char'. The default character is a space. (See input).
Example:

```
     10 PRMT(@)
     20 INPUT "GIMME A NUMBER" A0
     . . . .
```

will produce the output

```
     GIMME A NUMUER@
```

READ {(line-no)} var1 {, var2, var3, ...}

     This command is used to store values from DATA statements into the specified variables. The optional line number is used to reset the data pointer to a specific DATA statement. The RESTORE command is the more normal method of resetting the data pointer to a specified line, however.
Example:

```
     10 INPUT I
     20 READ (I*10+30) A0$ : REM this can't be renumbered
     30 PRINT A0$: GOTO 10
     40 DATA "MESSAGE NUMBER 1......."
     50 DATA "MESSAGE NUMBER 2......."
     60 DATA "MESSAGE NUMBER 3......."
     RUN

     ? 2 <cr>
     MESSAGE NUMBER 1......
     ? etc.
```

```
5 A1=0
10 FOR X=1 TO 10
20 READ A0
30 A1=A1+A0
40 NEXT X
50 PRINT "SUM OF DATA IS ";A1
100 DATA 1,2,3,4,5,6,7,8,9,1.2

RUN
SUM OF DATA IS 46.2
```

An attempt to read more data than available will result in an error. (See RESTORE.)

REM

Used as the last or only statement of a line to insert user comments in the program. All text between REM and the end of the line is ignored during execution.

Note that after a REM, a comment will stay in lower case automatically, making it much easier to read.

Do NOT try to put comments after DATA statements!

Example:

```
05000 REM This subroutine does exactly nothing
05010 RETURN
```

RENUM { new_start {,increment {,start {,finish} } } }

The renumber command is a complex program which will regularize the line numbering of a Basic program, changing all references to line numbers in the process. There are five forms of renum:

RENUM
This form will renumber the entire program so that when renumbered, the first line number will be 100 and the rest will increment by 10 from there on.

RENUM n
This form renumbers the entire program so that when renumbered, the first line number will be n and the rest will increment by 10.

RENUM n,i
This form renumbers the entire program so that when renumbered, the first line number will be n and the rest will increment by i.

RENUM n,i,s
This form renumbers the section of the program from the original line number s through to the end of the program, so that the line which was s will have line number n, incrementing by i.

RENUM n,i,s,f
This form renumbers the section of program between original line numbers sand f so that the line which was s will have line number n, incrementing by i.

Having given such a command, there are several reasons why it could fail. If the renumber program does detect an error condition, it will exit without having made any changes.

If the new linenumbers would either start before currently existing lines which are not being renumbered, or finish after current lines which are not being renumbered, a line number clash error will be given.

If the linenumbers exceed 65534, an illegal line error will be given.

If a renumbered line would turn out to be too long (>184 characters), a line too long error is given.

If the renumber program finds a label reference which refers to a non-existent line, the renumbering cannot proceed until this reference is fixed up.

This renumber command will fix all non-computed GOTO, GOSUB, ON...GOTO, ON...GOSUB, RESTORE, READ(lnum), THEN, ELSE and NEXT* references. Other commands such as LIST, DELETE, RENUM must be renumbered by hand if they are ever used.

Example:

BEFORE ...

```
00002 PRINT "this is a demo program"
00011 INPUT Q7
65023 PRINT [F8.4 Q7] :GOTO 2
```

AFTER a "RENUM 1,1" .. .

```
00001 PRINT "this is a demo program"
00002 INPUT Q7
00003 PRINT [F8.4 Q7] :GOTO 1
```

RESET{H}{[]} int-exp1,int-exp2

This command "turns off" a graphics point on the VDU screen. The command is the same for both HIRES, and LORES graphics modes, however the maximum co-ordinate values differ. (This command has nothing to do with the RESET key).

For LORES graphics, the int-exp1 must be in the range 0 to 127 and specifies a position across the screen, int-exp2 is in the range 0 to 47 and specifies a position down the screen. For HIRES graphics, int-exp1 must be in the range 0 to 511 across the screen to the right, and int-exp2 must be in the range 0 to 255 up the screen.

If an "H" is appended to the keyword RESET, then the Y axis

will be inverted automatically. If the "H" option is not used then a space MUST follow the RESET keyword.

Examples:
RESET 0,0 resets a point at the bottom left hand corner of the screen whereas RESET 63,24 resets a point near the centre of the screen (assumed LORES mode) .

RESTORE {int-exp}

The RESTORE command resets or repositions the position of the DATA pointer which controls the place in the program from where the next DATA item will be read. If the optional int-exp is added after the word RESTORE, the BASIC will set the DATA pointer to that line, otherwise the data pointer is set to the first piece of data in the program. (See also READ.)

Example:
```
00100 RESTORE 120
00110 DATA "skip this one"
00120 DATA "get this one instead"
```

RETURN

The RETURN statement is used to indicate the end of a subroutine to which a GOSUB call has been made. It causes execution to continue after the last GOSUB to be executed. If no GOSUB has been already executed when a RETURN is found, an error will occur. (See GOSUB.)

Example:
```
00100 GOSUB 200
00110 GOSUB 200
00120 END: REM If this is not here, an error will occur
00200 PRINT "*****************************************"
00210 RETURN
```

RUN

This command causes execution of the current BASIC program in memory to start at the lowest line number in the file. Before execution begins, all variables are cleared, all DIMensions are erased, the DATA pointer is reset to the beginning of the program, and the arithmetic stack is initialised.

SAVE{F} "file_name"

The SAVE command, a companion to LOAD, is the means whereby BASIC programs can be saved on cassette tape.
The default speed for cassette saving is 300 baud (bits per second). If the letter "f" (or "F") is added after the SAVE

keyword, the saving speed will be 1200bd. 1200bd is nearly four times faster than 300bd but generally requires a good tape recorder to achieve reliable results.

The filename is mandatory, and must be six or less alphanumeric (upper case, lower case and number) characters enclosed by double quotes.

Example:

```
>FOR A=1 TO 3 : SAVEF "TEST" : NEXT A
```

SD int-exp

Used to set the number of significant digits used in REAL calculations for greater precision or speed. SD can be set to any even number from 4 to 14 places. The default value is 8.

SD should only be used at the start of a program BEFORE any arrays are dimensioned or REAL variables used, and not changed from then on.

If you are using STRING ARRAYS in your program, then do not set SD at 4, a minimum of 6 should be used to ensure correct space allocation for the string array.

Example:

For drawing a circle on the screen, you would need quick, but not very accurate trigonometric functions, so use SD 4, but for solving equations where there is a possibility of cancellation errors, use SD 14.

SET{H} int-exp1,int-exp2

SET turns on a graphics dot in either HIRES or LORES graphics mode.

For LORES mode, int-exp1 must be in the range 0 to 127 and specifies a position across the screen, int-exp2 is in the range o to 47 and specifies a position UP the screen.

For HIRES mode int-expl must be 0 to 511 across and int-exp2 in the range 0 to 255 up the screen.

If an "H" character is put after the SET keyword, then the Y axis will be inverted. If no "H" is used then a space MUST follow the SET keyword.

Example: turn on every graphics dot the hard way

```
00100 LORES
00100 FOR X=0 to 127
00100  FOR Y=0 TO 47
00100   SET X,Y
00100  NEXT Y
00100 NEXT X
```

SPC int-exp

SPC(int-exp) is used to direct print to print int-exp spaces before the next item in the list. SPC is different from TAB(int-exp) because SPC(n) will always print n spaces no matter where the cursor is when it is invoked.
SPC must appear only in a print list.

Example:
```
00100 PRINT "LEFT";SPC(40);"RIGHT"
```

SPEED int-exp

Slows down the VDU output by introducing a delay between characters.
FORMAT: SPEED n where n=0 to 255 (0 is the fastest and the default set at a cold start.)

Example:
```
>SPEED 20:LIST:SPEED 0
```

STOP

Used in a program to terminate execution. The following message is printed:
STOP AT line-no

... where line-no is the line number where the STOP was encountered. This command, although performing roughly the same function as an END command , is normally used during program fault-finding. Execution can be restarted by using the CONT command.
For an example, see CONT.

STRS (int-exp)

STRS is used to set a limit on the MAXIMUM amount of memory that strings are allowed to use during the course of the program. Note that setting a too large string size may not allow enough room for variables and arrays.
The default value for STRS, set after a NEW, CLEAR or COLD START is 256.
Brackets MUST enclose the int-exp.
Example:
```
00100 STRS (5000) : REM A lot of strings are going to be used
00110 DIM A0(200)
00115 REM now set all elements to null strings
00120 FOR I=1 TO 200 : A0$(I)="" : NEXT I
```

TAB (int-exp)

TAB is used to direct PRINT to start at a particular point on a line. The argument must be an integer and if the required TAB has already been passed over, the program will ignore the integer argument.

TAB must be used in a print list.

TAB(0) is equivalent to TAB(255).

Example:

```
00100 PRINT TAB(12);"Super dooper program !!!!"
```

TRACE ON, TRACE OFF

When TRACE is turned ON the line number of each line executed is listed on the VDU between square [] brackets. TRACE OFF removes the facility after troubleshooting is over.

TRACE can be used in immediate mode or in a program.

Example:

```
00100 TRACE ON : REM trace execution of subroutine at 1000
00110 GOSUB 1000
00120 TRACE OFF
```

UNDERLINE

The underline commmand sets up the VDU to print underlined characters in all print statements after this one. If the previous display mode was a graphics one (PCG, HIRES or LORES), the screen will also be cleared first.

Example:

```
00100 PRINT "not underlined"
00110 UNDERLINE:PRINT "underlined":NORMAL
```

VAR (var1,var2,...)

The first statement of a subroutine to which arguments are passed. The variables in the VAR list receive their values from the calling GOSUB. The variables MUST correspond in position and mode to the expression in the GOSUB. These variables are common to the main program and thus can be used to pass values back to the main program.

If there are more variables in the VAR list than expressions in the calling GOSUB, the unused variables will retain their previous values. If there are too few variables in the list, an error condition will result.

Examples:

```
10 INPUT K,J
20 GOSUB [K*K,K+J] 100
. . . .
100 VAR (A,B)
110 PRINT A,B, A+B
120 RETURN
```

Suppose K is input as 10 and J is 20. In the subroutine, A will have the value 100 and B the value 30.

For advanced programs, it should be noted that the VAR statement can be used to simulate the PRINT IMAGE type of command found in other BASICs. Such statements permit the PRINT statement to be used with several different sets of variables.

```
10 INPUT P
20 GOSUB [P,"CATS"] 100
30 GOSUB [4*P, "CAT FEET"] 100
40 GOSUB [4*P*5, "CAT CLAWS"] 100
. . . .

100 VAR (D,D0$) :PRINT "MY HOUSE HAS ";D;D0$:RETURN
```

For p=6 this program will produce the following output:

```
MY HOUSE HAS 6 CATS
MY HOUSE HAS 24 CAT FEET
MY HOUSE HAS 120 CAT CLAWS
```

Note that this parameter passing mechanism does not create any "special variables. All variables used are normal, global (accessible everywhere) BASIC variables.

ZONE int-exp

This statement sets the ZONE width applied when commas are used in PRINT statements to 'int-exp' (See PRINT). The value of 'int-exp' may range from 1 to 16.

The default ZONE value, set by a 'NEW' or COLD RESET is 8.

Example:

```
00100 ZONE 12
00110 PRINT A,B,C,D,E,F
```

5.2 FUNCTIONS IN MICROWORLD LEVEL II BASIC

MICROWORLD BASIC was developed as an interpreter for use in business and game applications. The choice of functions reflects this interest. Functions are either real or integral and produce either real or integral results depending on the type. Integer functions should only appear in integer expressions and real functions only in real expressions.

Note: A function is different from a command in that it must always appear on the right hand side of an equation.
Example:

```
N  =  PEEK  (300)   Gets the byte from location 300 and
                    PUTS it into variable N.
A0  =  RND * 500    Generate a random number and put it
                    into variable A0.
PRINT  PEEK  (0)    This is an apparent exception which still
                    follows the rule (above): it means, get the
                    byte from location 0 and store it in a
                    temporary store, then PRINT the temporary
                    store.
```

REAL FUNCTIONS

ABS (real-exp)

Produces the absolute value of the real expression, if it is positive, does nothing. If -ve then returns same value but positive.

```
Example:  PRINT ABS(9);ABS(-9)
Result:   9. 9.
```

ATAN (real-exp)

This function returns the trigonometric arc-tangent of the real expression evaluated in RADIANS. (Accuracy is about 0.0001%.)

To convert RADIANS to DEGREES simply multiply the result by 57.29577951 (to the correct number of significant digits) .

```
Example:  PRINT ATAN (1)
Result:   0.7853981
```

In BASIC 5.00 and 5.10, the magnitude of the argument to the ATAN function must not be less than 0.1. If the magnitude of the argument is less than 0.1, then simply use the value of the argument itself, which is a good approximation to ATAN for such small numbers .. to assign A0=ATAN(X0)

```
IF X0 < 0.1 THEN LET A0=X0 ELSE LET A0=ATAN(X0)
```

COS (real-exp)

This function returns the trigonometric cosine of the real expression, assumed to be expressed in RADIANS. Accuracy is about 0.00000001 %.

```
Example:  PRINT COS(0)
Result:   1.

Example:  X=INT(20*COS(T0))
```

EXP (real-exp)

This function returns the value of e (2.718281828) raised to the 'real-exp' power. Accuracy is about .0001% for normal range. This is the equivalent to taking the NATURAL anti-logarithm of the expression.

```
Example:  PRINT EXP(2)
Result:   7.3890597
```

FLT (int-exp)

This function converts integer expressions into real numbers and is used to avoid "mixed mode" conflicts.

```
Example: A0=FLT(A)
```

FRACT (real-exp)

This function returns the fractional part of the real expression.

```
Example:
     PRINT FRACT(6.84)  produces   0.84

     PRINT FRACT(120) produces    0.0
```

In BASIC 5.00 and 5.10, the argument to FRACT must not have a "zero" after the decimal point as in 2345.01653, or the resulting value will be unusable in arithmetic operations.

To avoid this problem, use the following construction to replace FRACT ..

instead of A0=FRACT(X0), use

```
   A0=X0-FLT(INT(X0))   as long as -32767 < X0 < +32766
```

FRE(0)

This REAL function returns a number representing the total memory available for program and variable storage. If you are running with 48K of RAM this number will come out negative! Don't

worry, this is only a restriction caused by the method used to represent integer numbers. In the immediate mode use PRINT FRE(0) to give an immediate indication of memory left at any particular point in time.

Example: (16k MicroBee after a NEW)
PRINT FRE(0) produces 13822.

FRE($)

Gives the maximum amount of "string space" still available as a real number. Use STRS (int-exp) to allow more "string space". The use of this function also "purges" the string space removing old strings which are no longer used.

Example: (After a NEW)
PRINT FRE($) produces 256.

LOG (real-exp)

This function returns the common logarithm of the real expression. The natural logarithm (to base e) can be found by using 2.30258*LOG (real-exp). (Accuracy is 0.0001%.)

ANTILOGS can be calculated by l0^(real-exp).

Example:
PRINT LOG(1000) produces 3.0000001

RND

The random number generator, returns a real number between 0 and 1. Note that there is NO argument allowed to RND.
To get other sized random numbers see examples below:

RND*150 Returns a number between 0 and 150

RND*200-l00 Returns a number between -100 and +100

INT(RND*6)+1 Returns an integer between 1 and 6

SGN (real-exp)

This function returns one of three values as follows:

-1 if real-exp <0
 0 if real-exp =0
+1 if real-exp >0

Example:
PRINT SGN(-12);SGN(0);SGN(7)

produces -1. 0 1.

SIN (real-exp)

This function returns the trigonometric sine of the real expression considered to be in radians. (Accuracy is about 0.00000001%.)

Example:
PRINT SIN(355/113) produces -3.0E-07 (close to 0).

SQR (real-exp)

Produces the positive square root of the value of the real expression. Note in this particular case the real-exp must be a positive number.

Example:
PRINT SQR(9) produces 3.

VAL (str-exp)

Converts a string expression like "1234.45" which contains the representation of a floating point number into the corresponding REAL number. VAL is always a REAL function, so if it is desired to convert a string which looks like an integer into an integer, the INT(VAL(str-expM-; construction must be used.
If the string expression cannot be converted into a REAL number, the value 0 is returned.

Examples:
A0=VAL("23.03") assigns A0 the value 23.03
A0=VAL("Apple []") assigns A0 the value 0.

INTEGER FUNCTIONS

ASC (str-exp)

ASC takes the first character of the string expression given, and returns the ASCII code for it (See appendix for list of ASCII codes).

Example:
PRINT ASC("1234") produces 49, ASCII code for "0"

ERRORC

ERRORC, a function of no parameters, returns the code number of the last error which occurred (see Section 3.15 for error

message list with code numbers).

ERRORL

ERRORL returns the line number at which the last error occured. ERRORL requires no parameters.

INT (real_exp)

Converts the value of the real expression into an integer.

Example:
```
SET INT(X0),INT(Y0)   gets around "mixed mode" troubles.
```

IN (int_exp)

Inputs a data byte from the input port with the address given by the value of the integer expression (in the range 0 to 255).

Example:
```
PRINT IN(0)    will print the data on parallel PORT A
```

LEN (str_exp)

Returns the length, or number of characters contained in the specified string expression.

Example:
```
PRINT LEN("fred"+"jane")      produces 8
PRINT LEN("")                 produces 0
```

PEEK (int_exp)

Reads the data byte stored in memory location addressed by the value of the integer expression.

Example:
```
PRINT PEEK(16*4096)      produces the character code of the
first character on the screen.
```

POINT{H}{[]} (int-exp1,int-exp2)

POINT returns a value depending on whether the specified dot is set or not.

If the co-ordinates are out of range for the relevant graphics mode, the value returned in -1.

If the specified point is set, POINT returns -1.

If the co-ordinates are in range, and the dot is not set,

POINT returns 0.
     The values of 0 and -1 were chosen to correspond to the two boolean value for an integer which can be used in an IF statement directly, so

```
10   IF POINT(X,Y) THEN LET A=-A:B=-B ELSE SET X,Y
```

 ... will negate A and B if the dot is set, and if it was not set, this statement will set it.
     When an "H" is appended to the POINT keyword, the Y axis is inverted. If no "H" is used, then a space must follow the keyword.

POS

     This integer function of no arguments returns an integer representating (theoretically) the cursor position on either the VDU or the line printer, it makes no allowance for the actual length of lines.

Example:
PRINT 4,4,4,POS produces 4. 4. 4. 23

SEARCH (str=exp1,str-exp2 {,int-exp})

     String str-exp1 is searched for <int-exp> th occurrence of substring str-exp2. The integer value returned is the position of the beginning of the substring, if found, or zero if not found. The default value of int-exp is 1.
Examples always help to illustrate the point:

```
     A0$ = "HOW ARE YOU"      define the string
     PRINT SEARCH(A0$,"ARE")  find 1st occurrence of "are"
     5                        it occurs from character 5 onwards.
     PRINT SEARCH(A0$,"O",2)  find the second "O" in A0$
     10                       it occurs in the 10th character
```

USED

     This function of no arguments returns the number of PCG characters used when in HIRES graphics mode.
     This information is very usedful because when the graphics subroutines attempt to use more than 128 PCG programmable characters to draw the HIRES graphics, an error is generated which aborts the program.

Example:

```
00100 HIRES
00110 PLOT 0,0 TO INT(RND*512),INT(RND*256)
00120 IF USED <60 THEN 110 ELSE 100
```

NOTE: Integer functions may be used directly in print statements as the type is implicit to the function.

STRING FUNCTIONS

CHR{$} (int-exp)

CHR$ performs the inverse of the ASC function: it returns a one character string whose character has the specified ASCII or graphics code. The value of int-exp may be any number from 0 to 255 except 128. The dollar sign after the CHR is optional.
Example:
PRINT CHR$(34) will print a double quote (which is not normally obtainable because" is used to delimit strings.)

CHR$ may also be used to display the LORES graphics characters ... (See LORES for example.)

CHR$ is also a good way of printing control characters which cannot be entered from the keyboard.
Example:

```
PRINT CHR$(7) : REM rings the bell
PRINT CHR$(13): REM RETURNs cursor without linefeed
```

KEY{$}

KEY$ returns a string depending on whether or not a key on the keyboard has been pressed.
If no key has been pressed since the last KEY$ call (or line input), the null string (length 0) is returned.
If a key has been pressed, the one-character string corresponding to that key is returned.
The characters which are received using the KEY$ function are NOT automatically displayed on the screen, allowing such things as "TURTLE GRAPHICS" and real time games which do not require the RETURN key to pressed and don’t stop when waiting for a key.
Because the time taken to look at the keyboard is very short, the KEY$ function is usually examined in some sort of loop.

Example:

```
00100 A0$="" : REM clear string to accumulate
00110 A1$=KEY$ : IF A1$="" THEN 110 : REM get key
00120 A0$=A0$+A1$ : REM add this key to A0$
00130 IF ASC(A1$)<> 32 THEN 100 : REM wait for a space
00140 PRINT A0$ : REM view captured string
```

STR{$} (num-exp)

STR$ converts an integer or real expression into a string - the opposite of VAL (str-exp) .

For example, suppose A0=1234.56, STR$(A0) gives the string " 1234.56" just as if A0 had been printed.

5.3 USER DEFINED FUNCTIONS

MICROWORLD BASIC also has provision for the user to define special functions for himself. The basis is the FN statement.

```
FNn = expr
```

The format is FNn where n is an integer between 1 and 7. If when defining an expression a dummy variable is required, a '#' sign should be used to indicate where the dummy variable should appear. Example:

```
10 FN2 = 3.14159 * # + 1.8
```

When a reference is made to the defined expression in a program, it appears as FNn(expr). The value of the argument would be passed into each '#' symbol of the defining statement.

It is important that the mode (real or integer) of the argument be the same as that of the result type. Also, the result type of a user defined function is assumed to be REAL unless it is used in some construction which implies integer mode (such as LET A=... or TAB(...).

Example:

```
     10 FN0 = #+#
     20 A=4
...
     40 PRINT FN0(6)
     50 B=FN0 (A)
```

This program will print the value 12. and assign 8 to the integer variable B.
If we had line 60 as

```
     60 PRINT FN0(A)
```

then a MIXED MODE error would occur.

Some special functions:

FNn = SIN(#)/COS(#) This real function calculates the trigonometric tangent of an angle. the angle must be expressed in RADIANS.

FNn = ATAN(#/SQR(l-#*#)) This real function calculates the inverse trigonometric sine (ARC SINE).

MACHINE LANGUAGE SUBROUTINES

USR (int-exp1 {,int-exp2})

This integer FUNCTION (NOT A COMMAND) produces a call to a machine language routine given by the integer expression int-exp1. The value of int-exp2 is passed in the BC register pair. The int-exp2 is optional, in which case BC will contain zero.

The machine language program may use all registers, but the stack must be managed so that PUSH's and POP's (if that's how you say it) are equal in number. In this case a machine language return instruction will produce reentry back to MICROWORLD BASIC. To pass a value back it should be placed in the Band C registers before return. This integer value becomes the "value" of the function.

USR must be the only expression on a line of BASIC otherwise errors will occur.

To communicate a large amount of data to a machine language routine, use a POKE instruction to put the data into a known area of memory (called a 'chat' area) before the USR function is used. Then use the PEEK instruction to get it back again.

Example:

```
10 I = USR(0,55)
```

passes the value 55 (decimal of course) in the BC pair to the subroutine assumed to have been previously poked into memory at zero, then puts the returned value of the BC pair into integer variable I after the subroutine has returned.

Section 6.

Application Programs

SECTION 6: APPLICATION PROGRAMS IN MICROWORLD BASIC

Listed below are applications programs written in MICROWORLD BASIC. They are included, not as examples of high quality programming (they are certainly not!), but as illustrative examples to further enhance your comprehensive understanding of some of the more common techniques in BASIC.

6.1 GUESSING GAME

Illustrates the use of input statements, conditional branching, random number generation, and indexed looping. This should provide enough raw material for you to write your own versions of this type of game.

```
100  REM *** NUMBER GUESSING ***
110  REM A SIMPLE PROGRAM IN MICROWORLD BASIC
115  CLS: SPEED 50
116  PRINT:PRINT:PRINT
120  PRINT "PLAYER VS THE COMPUTER GUESSING GAME"
125  T=0
130  PRINT "I'M THINKING OF A INTEGER BETWEEN 0 AND 100"
135  PRMT(?)
140  N=INT(RND*100)
141  PRINT N
142  REM REMOVE LINES 141,142 BEFORE PLAYING THE GAME
150  INPUT "WHAT IS YOUR GUESS " G
155  T=T+l
160  IF G=N : GOTO 1000
170  IF G<N : GOTO 500
180  PRINT "TOO BIG - TRY AGAIN"
190  GOTO 150
500  PRINT "TOO SMALL - TRY AGAIN"
510  GOTO 150
1000 PRINT "RIGHT ON - AFTER ";T;" TRIES."
1010 GOTO 125
1020 END
```

6.2 A SORTING ROUTINE: RIPPLE SORTING

This is a simplified example of the traditional ripple sorting algorithm. It also provides a useful example of the READ and DATA method of assigning values to variables and also dimensioning a matrix.

```
100 REM *** RIPPLE SORT: MAXIMUM 100 NUMBERS ***
110 CLS
200 DIM A1(100)
210 DATA 5
220 DATA 6.2, 15.8, 23.0, 12.9, 11.1

230  REM CONTINUE DATA STATEMENTS AS REQUIRED
1000 READ N          :REM THIS READS DATA IN LINE 210
1010                 REM IN OUR CASE 5 ENTRIES TO SORT
```

```
1020 FOR 1=1 TO N
1030 READ A1(I)
1040 NEXT I
1050 REM
1060 REM
1070 FOR S=1 TO N-1
1080 LET M=0
1090 FOR I=1 TO N-S
1100 IF A1(I) <= A1(I+1) THEN 1500
1110 LET X1=A1(I)
1120 LET A1(I) = A1(I+1)
1130 LET A1(I+1) = X1
1140 LET M=1
1500 NEXT I
1510 IF M=0 THEN NEXT*S 2000
1520 NEXT S
2000 FOR I=1 TO N
2010 PRINT A1(I)
2020 NEXT I
2030 END
```

6.3 ANNUITIES AND COMPOUND AMOUNTS

BASIC has widespread uses in business applications and is the major language used in many small business computer systems. The simple programs below illustrate just how easy it is to apply MICROWORLD BASIC to solving complex problems.

```
100 REM ** THIS PROGRAM CALCULATES THE FUTURE VALUE **
110 REM Y1=PRESENT VALUE, I1=INTEREST RATE
120 REM N1=TIME PERIODS, X1=FUTURE VALUE
130 INPUT "ENTER PRESENT VALUE " Y1
140 INPUT "ENTER INTEREST RATE PER PERIOD " I1
150 INPUT "ENTER NUMBER OF PERIODS " N1
160 X1=Y1*(1+I1)^N1
170 PRINT "AFTER ";N1; " PERIODS ";Y1;"IS WORTH "; X1
180 INPUT "DO YOU WANT TO KNOW MORE, Y OR N ?" A1$
190 IF A1$ = "y" THEN GOTO 130
200 PRINT "BYE FOR NOW"
210 END
```

This program calculates the amount of money you would have to invest to have a given sum at the end of a period of time.

```
100 REM ** CALCULATES PRESENT VALUE BASED ON FUTURE SUM
105 CLS:PRINT:PRINT:PRINT:PRINT
110 INPUT "FUTURE VALUE " X1
120 INPUT "INTEREST RATE PER PERIOD" I1
130 INPUT "NUMBER OF YEARS " N1
140 Y1=X1*(1+I1)^(-N1)
```

```
150 PRINT"TO HAVE";X1;"IN";N1;" YEARS, YOU WILL NEED"
160 PRINT "TO INVEST ";Y1;" NOW"
170 INPUT "DO YOU WANT MORE? (ANSWER YES OR NO)" A1$
180 IF A1$="YES" THEN 110
190 PRINT "BYE FOR NOW"
200 END
```

6.4 DEGREES TO RADIANS

MICROWORLD BASIC uses radians in all trigonometric functions. Often you will want to convert values expressed in DEGREES to RADIANS and this short subroutine (see the RETURN statement?) will no doubt prove very useful.

```
100 REM ** THIS PROGRAM CONVERTS DEGREES TO RADIANS **
110 CLS:PRINT:PRINT:PRINT
120 INPUT "ENTER DEGREES TO BE CONVERTED" D1
130 LET R1=3.14159*D1/180
140 PRINT D1; " DEGREES CONVERTS TO ";R1; "RADIANS"
150 RETURN
```

6.5 ELECTRONICS APPLICATIONS: OHMS LAW CALCULATOR

By now you may have noticed how easy it is to actually apply BASIC to problem solving. The key is to find a function that will solve some equation, input the known variables and calculate the unknowns. This program illustrates the well known "OHMS LAW" used universally in electronics.

```
100 REM ** THIS PROGRAM DEMONSTRATES OHMS LAW **
110 CLS:PRINT:PRINT:PRINT
120 PRMT(?)
125 INPUT "DO YOU WANT TO CALCULATE 'V', 'A' OR 'R'" S1$
130 IF S1$="V" THEN 200
135 IP S1$="A" THEN 250
140 INPUT "ENTER THE KNOWN VOLTAGE "V0
150 INPUT "ENTER THE KNOWN CURRENT "A0
160 R0=V0/A0
165 PRINT "YOU WILL GET A VOLTAGE DROP OF ";V0;
170 PRINT "WHEN ";A0;" AMPS PASSES THROUGH ";R0;" OHMS"
175 GOTO 110
200 INPUT " ENTER THE RESISTANCE "R0
205 INPUT " ENTER THE CURRENT "A0
210 V0=R0*A0
215 PRINT V0;" VOLTS"
220 GOTO 110
250 INPUT "ENTER THE RESISTANCE "R0
255 INPUT "ENTER THE VOLTAGE "V0
```

```
260 I0=V0/R0
265 PRINT "THE CURRENT IS ";I0;" AMPS"
270 GOTO 110
300 END
```

6.6 GRAPHICS WITH MICROWORLD BASIC

MICROWORLD BASIC enables you to "draw" on the VDU screen using the SET x,y command, erase from the VDU using the RESET x,y and to test if a particular point is actually set (POINT (x,yM-;. High resolution is also possible. This simple program will provide a useful basis for displays and games.

```
90 REM ** THIS PROGRAM DEMONSTRATES GRAPHICS
95 REM AND THE RANDOM FUNCTION **
100 CLS:LORES
115 FOR A=1 TO 123:SET A,31:NEXT A
120 X=INT(RND*123)
130 Y=INT(RND*47)
140 SET X,Y
150 IF X=60 OR X=120 THEN 245
155 FOR A=1 TO 123 :RESET A,31:NEXT A
160 GOTO 120
245 FOR B=1 TO 47 :SET 62,B: NEXT B
255 X=INT(RND*123)
260 Y=INT(RND*47)
270 RESET X,Y
280 IF X=120 THEN 115
290 FOR B=1 TO 47: RESET 62,B : NEXT B
300 GOTO 255
310 END
```

6.7 MUSIC ON THE MICROBEE

The MicroBee can be easily programmed to play music through the internal loudspeaker. Although you are restricted to monophonic reproduction the results can really be amazing. Try typing in the following program and after you have run it answer the question below.

```
100 PLAY 4,2; 8,2; 9,2; 11,10; 4,2; 8,2; 9,2; 11,10
110 PLAY 4,2; 8,2; 9,2; 11,4; 8,4; 4,2; 8,4; 6,10
120 PLAY 8,2; 8,2; 6,2; 4,8; 8,4; 11,4; 11,2; 9,10
130 PLAY 8,2; 9,2; 11,4; 8,4; 4,4; 6,4; 4,10
```

What is the tune? .....

6.8 PCG CAR GRAPHICS

This program is the final result of the development process described in section 3.10 of defining and using programmable characters to create a picture of a car.

```
  100  P=63488+65*16: REM pcg address of "A"
  110  FOR A=P TO P+16*3-1 : REM putting in 3 chars
  120  READ B : POKE A,B
  130  NEXT A
  140  PRINT "Look ...":GOSUB [2] 2000:PRINT " +";:GOSUB [3]
       2000: PRINT" =";: GOSUB [5] 2000
  150  END
 1000  DATA 0,0,0,0,0,7,8,112,128,128,248,7,0,0,0,0
 1010  DATA 0,0,0,0,0,254,1,0,0,0,124,131,0,0,0,0
 1020  DATA 0,0,0,0,0,0,128,120,6,1,63,192,0,0,0,0
 2000  VAR (R) : REM receive no. of cars to print
 2010  FOR I=1 TO R
 2020  PRINT" ";:PCG: PRINT "ABC";: NORMAL
 2030  NEXT I
 2040  RETURN
```

CONVERTING FROM OTHER BASICS

How many times have you seen a useful routine written in another version of BASIC and wanted to use it on your own computer? The MicroBee is equipped with a powerful editor, error checking routines and the 'GX' (Global Search and Replace). Let's see how easy it is to convert from another BASIC into MICROWORLD BASIC.

The first step is to type in the program or better still, use the RS232 interface to read in the program text from the other computer. Let's assume the following program.

```
  100  REM A DEMO PROGRAM IN BANANA HARD BASIC
  110  HOME
  120  A$="BILL LIVES IN A BRICK HOME"
  130  B$=LEFT$(A$,4)
  140  PRINT "HELLO "+B$
  150  END
```

Type RUN<CR>
and the MicroBee will respond with

```
  Syntax error in line 00110
  00100 HOME (cursor indicated by underscore)
```

Well MICROWORLD BASIC uses CLS to clear the screen not HOME, so let's replace it as follows.

Type GX/HOME/CLS/<CR>
and the MICROBEE will respond with

00110 HOME you press the period (full stop key) to replace this line and then the MICROBEE will respond with
00120 A$="BILL LIVES IN A BRICK HOME" you press the SPACE

bar because you do not want to replace HOME in this string.

Now type LIST<CR>
and the MICROBEE will respond with

```
     00110 REM etc
     00110 CLS
     00120 A$="BILL LIVES IN A BRICK HOME"
     00130 B$=LEFT$(A$,4)
     00140 PRINT "HELLO "+B$
     00150 END
```

and type RUN<CR>
and the MICROBEE will respond with

```
     Integer string error in line 120
     00120 A$="BILL LIVES IN A BRICK HOME"
```

well MICROWORLD BASIC requires that all strings be of form A1$, M4$, J7$ etc so just type

```
     GX/A$/A1$/<CR>
```

and press the period key to replace A$ with A1$ as we did above. While you are at it you should change line 130 to read B1$ as well. Let's type EDIT 130 and the MicroBee will respond with

```
     00130 B$=LEFT$(A1$,4)
```

hold one finger on the CONTROL key and press S until the cursor is over the '$' then press 1 <CR> and the line should read

```
     00130 B1$=LEFT$(A1$,4)
```

and also change B$ in line 140 to B1$ now press RUN<CR>

What another error??

```
     NEXT without FOR error in line 00130
     00130 B1$=LEFT$(A1$,4)
```

Oh well the error messages can sometimes be 'fooled' by certain words. In reality MICROWORLD BASIC will not recognize the LEFT$ construction so we will have to replace it. Refer to Section 3 of this manual and you will note that we have to replace

```
     LEFT$(A1$,4) with A1$(;1,4)
```

so we use the following

```
     GX/LEFT$(A1$,4)/Al$(;1,4)/ <CR> and press '.'.
```

Type run and there you have it. Given that you should by now be familiar with the DOs and DON'Ts of MICROWORLD BASIC, this approach will allow you to convert from any other BASIC by just using the error messages to find out what MICROWORLD BASIC doesn't like, EDITing with the built in editor and replacing with the GX command where it is convenient.

Section 7.

Glossary

SECTION 7:GLOSSARY

7.1 A GLOSSARY OF PERSONAL COMPUTER TERMS.

This glossary will help you to understand frequently used computer terms.

ADDRESS. A number which identifies the specific location where a piece of information is stored in the memory of the computer.

ALPHANUMERIC. Characters consisting of letters and numerals, as opposed to special characters.

ASCII. American Standard Code for Information Interchange. Computers use binary numbers to represent letters, numerals, and special characters.The ASCII code specifies which binary number will stand for each character.

ASSEMBLY LANGUAGE. a means of communicating with a computer at a low level. Assembly language lies between high level languages, like MICROWORLD BASIC, and machine language, ones and zeros. Programmers use assembly language to make efficient use of memory space and to create a program which runs quickly.

BASIC. Beginners All-purpose Symbolic Instruction Code.The most used high-level language for small computers.

BAUD. A measure of the speed at which computer information travels. A baud is equal to one bit per second. MICROBEE users 300 or 1200 baud.

BINARY NUMBERS. A numbering system that uses only ones and zeros.lt is an efficient way of storing information in a computer because the hundreds of thousands of microscopic switches in a computer can only be on (1) or off (0).

BIT. A binary digit (1 or 0) ,the smallest item of useful information that a computer can handle.

BUG. An error. A hardware bug is a malfunction in the computer. A software bug is a programming error.

BYTE. A sequence of bits that represent a single character. In most small computers, a byte is eight bits.

CAI. Computer-Aided Instruction. Teaching by means of a computer.The computer informs the student of right and wrong answers as he or she makes them.

CHARACTER. A single letter, number, or other symbol.

CHIP. A generic term for an integrated circuit (IC), a single package holding hundreds of thousands of microscopic components. The term comes from the slices (chips) of silicon which they are

made of.

COMMAND. A word or character that causes a computer to do something.

COMPUTER. Any device that can receive and then follow instructions to manipulate information. In any computer, both the information on which the instructions operate may be varied from one moment to another. A device whose instructions cannot be changed is not a computer.

COMPUTER NETWORK. Two or more connected computers that have the ability to exchange information.

COMPUTER PROGRAM. A series of commands, instructions, or statements put together in a way that tells a computer to do a specific thing or series of things.

CONTROL CHARACTERS. Characters or commands obtained by holding down the key marked "CTRL" while pressing another key.

CP/M. Control Program/Microcomputer.

CPU. Central processing unit, the heart of a computer. The CPU controls all operations of all parts of the computer and does actual calculations. In personal computers, CPU usually refers to just one of the chips in the machine.

CURSOR. A position indicator on a VDU. On your MICROBEE it is >

DATA. A general term meaning any and all information, facts, numbers, letters, and symbols which can be acted on or produced by a computer.

DATA BASE. A collection of related data that can be retrieved by a computer.

DEBUG. To go through a program to remove mistakes.

DISASSEMBLER. A program that translates a computer’s native language into assembly language.

DISK. A round piece of magnetic-coated material,either rigid metal or flexible (floppy) plastic, used to store data with greater density, speed, and reliability than is available on cassettes.

DISPLAY. A method of representing computer information in visual form. Most common are VDU and printed paper.

DOCUMENTATION. (1) The instruction manual for a piece of hardware or software.(2) The process of gathering information while writing a computer program so that others using the program are able to see what was done.

DOS. Disk Operating System.

FIRMWARE. A term refering to software that has been permanently placed in memory, usually into ROM (Read Only Memory) .

FLOPPY DISK. A thin, flexible disk of plastic with a magnetic coating used for data storage.

FLOWCHART. A common method of graphically planning what a piece of software should do before the actual writing process begins, or for describing what it does after it is written.

HARD COPY. A paper printout of information produced by the computer.

HARDWARE. The physical part of the computer (VDU ,CPU) as opposed to software.

HEXADECIMAL NUMBERS. A number system with the base of 16 commonly used by programmers to indicate locations and contents of a computer’s memory.

INITIALISE. To prepare a disk so that the computer can later store data on it.

INPUT. The transfer of data into the computer.

INPUT/OUTPUT. Called I/O for short. A general term for 1) external equipment connected to a computer. 2) Two-way exchange of information that goes on between the computer and that equipment.

INTEGRATED CIRCUIT (IC). Also known as a chip, this is a group of interrelated circuits in a single package.

INTERFACE. A piece of hardware or software used to connect two devices that cannot be directly hooked together.

LANGUAGE. A set of conventions (symbols and terms) specifying how to tell a computer what to do.

LOAD. To put data and/or programs into a computer.

MACHINE LANGUAGE. The "native language" of a computer; those fundamental instructions the machine is capable of recognising and executing. .

MEMORY. Circuitry and devices that hold information in the form of binary ones and zeros that the computer can access. Examples are main memory (integrated circuits), floppy disks, and cassette tape.

MENU. A list of commands that most ready-made programs will display on request.

MICROCOMPUTER. A computer based on a microprocessor like MICROBEE.

MICROPROCESSOR. The central processing unit of a computer (usually in a single intergrated circuit) which holds all the elements for manipulating data and performing arithmetic calculations.

MODEM. MOdulator-DEModulator. This device allows a computer to communicate over phone lines.

MONITOR. A television set, often one that is specially manufactured to be connected to a computer.

NETWORK. An interconnected system of computers and/or terminals, often connected by telephone lines.

OPERATING SYSTEM. Software that oversees the overall operation of a computer system. This group of programs acts as an intermediary between the hardware and the applications software.

PERIPHERAL. A piece of equipment (usually hardware) that is external to the computer itself. i.e. Disk drives and printers.

PERSONAL COMPUTER. A general purpose inexpensive computer owned by an individual.

PRINTER. An output device that produces a printed ("hard") copy of the information generated by the computer. A line printer prints a whole line of text at a time. A serial printer prints one character at a time.

PRINTOUT. A printed copy of the information produced by the computer.

PROGRAM. 1) A set of instructions that tell the computer to do something. 2) To prepare the set of instructions.

RAM. Random Access Memory. The main type of memory used in a small computer. Also known as read/write memory because data in RAM can be easily changed.

RF MODULATOR. A device that lets a personal computer use any ordinary television set for output.

ROM. Read Only Memory. Memory where information is permanently stored and cannot be altered. This form of memory is also random access.

SAVE. To store a program on a disk or cassette.

SIMULATION. A computerised representation of something in action.

SOFTWARE. Programs or segments of programs. The term was coined to contrast with hardware.

SYSTEM. An organised collection of hardware and software which works together.

SYSTEM SOFTWARE. General purpose programs that allow programmers to modify applications programs. BASIC may be considered part of the system software.

TERMINAL. A piece of equipment with a keyboard for input and an output device such as a VDU or printer. A terminal is used to communicate with the computer.

USERS' GROUP. An association of people who exchange information about a particular computer. There is a MICROBEE Users' Group and there is a Newsletter that circulates monthly named "Micro World News".

VDU. Visual Display Unit. A TV monitor which displays output.

WORD PROCESSING. The entry, manipulation, editing, and storage of text using a computer.

Section 8.

Index
Appendix
Updates

8.2 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	SA	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	-	SF	95
LF	0A	10	5	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
DCl	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	lA	26	E	45	69	p	70	112
ESC	IB	27	F	46	70	q	71	113
FS	lC	28	G	47	71	r	72	114
GS	ID	29	H	48	72	s	73	115
RS	lE	30	I	49	73	t	74	116
US	IF	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			

8.3 Important Memory Locations in MicroBee BASIC

Note that all 16 bit numbers stored have the Least Significant Byte stored first, followed by the Most significant Byte.

Decimal address	Hex address	Function
0	0	HIRES scratch
128	80	INT vectors
136	88	PIO INT vectors
160	A0	Top of memory pointer
162	A2	Warm start jump address
164	A4	Init. check bytes
166	A6	Mach. lang. EXEC address
168	A8	10 reserved bytes
178	B2	Output device vector table
194	C2	Input device vector table
210	D2	Start of 6545 reg. table
212	D4	6545 Horizontal
217	D9	6545 Vertical
220	DC	6545 Cursor start/control
226	E2	Output device byte
227	E3	List output device byte
228	E4	Input device byte
229	E5	Video mode byte
230	E6	Output speed byte
231	E7	PCG chars USED
232	E8	plot type I/R/S
233	E9	Tape speed 1(1200)/4(300)
234	EA	RS232 baud 0(300)/1(1200)
235	EB	PLOT/ tape buffer
512	200	Low ML start if auto-exec
2240	8C0	No. sig. digits (must be <62)
2256	8D0	Prog. begin pointer
2258	8D2	prog. end ptr (for recovery)
2304	900	Start of BASIC program/ML
?	?	End program, start variables
16128	3F00	Stack/top of strings for 16k
16383	3FFF	End of RAM in 16K system
32512	7F00	Stack/top of strings for 32k
32767	7FFF	End of RAM in 32K system
32768	8000	BASIC, warm,cold start
32771	8003	Cold,warm start
32774	8006	DGOS wait keyboard in A
32777	8009	DGOS key if available -> NZ
32780	800C	DGOS vdu out in B
32783	800F	DGOS give PIO an arm
32783	8012	DGOS cass byte in A
32789	8015	DGOS cass block in
32792	8018	DGOS cass byte out A
32795	801B	DGOS cass block out
32798	801E	RUN0 for power on execute
32801	8021	Warm for restoring ROJ

32804	8024	HIRES init
32807	8027	LORES init
32810	802A	INVERSE init
32813	802D	UNDERLINE init
32816	8030	SET dot X=HL, Y=DE ..
32819	8033	RESET dot returns Z if O.K.
32822	8036	INVERT dot
32825	8039	TEST for dot-NZ if set/error
32828	803C	PLOT a line
32831	803F	Redirected input A
32834	8042	Redirected out A
32837	8045	Redirected print out A
49151	BFFF	End of BASIC roms
49152	C000	EDASM if fit ted
49155	C003	EDASM monitor entry point
57344	E000	NET/MEM ROM if fitted
61440	F000	Start of screen memory
63488	F800	Start of PCG ram

For more details, ask about the availability of the BASIC SCRATCH and JUMP TABLE sheets.

8.4 Full MicroBee PORT MAP

Some of these PORTS are for DISK/ S100 expansion users' reference only.

PORT	FUNCTION
00	PIO port A data port
01	PIO port A control port
02	PIO port B data port
03	PIO port B control port
08	Colour selection port (colour only)
09	2651 USART port (colour only)
0A	S100 Extended addressing port, Mem selection
0B	ROMREAD latch on bit 0 (to read char gen)
0C	6545 CRTC address/status port
0D	6545 CRTC data port
44	FDC command/status
45	FDC track register
46	FDC sector register
47	FDC data register
48	Controller select/side/double density latch

PORT B data port bit assignment:

bit 0 cassette data in
bit 1 cassette data out
bit 2 RS232 CLOCK or DTR line
bit 3 RS232 CTS line
bit 4 RS232 input
bit 5 RS232 output

bit 6 speaker bit
bit 7 network interrupt bit

8.5 MICROWORLD BASIC TOKEN CODES

In MicroWorld BASIC, all the reserved words are compressed into one byte to save memory spaceo The codes used are as follows.

129	LET	162	PRMT	195	ERROR
130	LPRINT	163	ZONE	196	POS
131	PRINT	164	SD	197	ASC
132	IF	165	CLEAR	198	USED
133	NEW	166	EDIT	199	NET
134	LLIST	167	SET	200	MEM
135	LIST	168	RESET	201	EDASM
136	ELSE	169	SPEED	202	GX
137	THEN	170	NORMAL	203	ABS
138	FOR	171	UNDERLINE	204	RND
139	NEXT	172	SAVE	205	FLT
140	DIM	173	LOAD	206	FRE
141	GOTO	174	STRS	207	VAL
142	OFF	175	INVERSE	208	FRACT
143	ON	176	PCG	209	SGN
144	STOP	177	CURS	210	SQR
145	END	178	NOT	211	SIN
146	GOSUB	179	AND	212	COS
147	READ	180	OR	213	ATAN
148	DATA	181	TRACE	214	LOG
149	RETURN	182	CONT	215	EXP
150	INPUT	183	CLS	216	PLOT
151	RUN	184	HIRES	217	DELETE
152	RESTORE	185	AUTO	218	RENUM
153	TO	186	INVERT	219	PLAY
154	STEP	187	LORES	220	EXEC
155	TAB	188	INT	221	STR
156	SPC	189	IN	222	KEY
157	FN	190	PEEK	223	CHR
158	VAR	191	USR		
159	POKE	192	LEN		
160	OUT	193	SEARCH		
161	REM	194	POINT		