Extended BASIC
Versions 5 & 6

for Stand-alone Disc Systems and Network Stations

Reference Manual

PN 11006, Revision 2

The policy of Research Machines Ltd is one of continuous development and improvement of its products and services, and the right is therefore reserved to revise this document or to make changes in the computer software it describes without notice. Research Machines endeavour to ensure that the products described perform correctly according to their descriptions. However, Research Machines Limited do not accept liability for the consequences of any error or omission.

The original labelled distribution disc is regarded as the only proof of purchase and must be produced in order to qualify for an update at a reduced rate. Keep it safe and always work from copies.

Additional copies of this publication may be ordered from Research Machines at the address above. Please ask for "Extended BASIC Versions 5 and 6 for Stand-alone Disc Systems and Network Stations, Reference Manual, PN 11006".

If you would like to comment on any of our products or services please use the reply paid form provided at the end of this manual.

RELEASE NOTE

In order to keep it up to date, the Release Note is not bound in with the manual but is supplied both as a printed sheet and as file RELEASE.PRN on the Distribution disc (the disc version probably being in a condensed form). The Release Note contains details of the memory addresses used by BASIC together with information about any improvements and additions since the Manual was published. To display the Release Note on the screen, follow the instructions in Chapter 2 for loading BASIC but instead of typing "BASIC" in response to the "B>" prompt, type:

```
TYPE RELEASE.PRN<RETURN>
```

If you wish to print it as well as displaying it, select a printer option first, as described in Appendix D, and type <CTRL/P> (hold down the CTRL key while typing the letter P) prior to pressing <RETURN>.

The text of this manual was produced
with the aid of word-processing software on a
Research Machines 380Z computer, and transmitted to
The Hazell Press of Wembley for photo-typesetting
in Century Schoolbook and OCR-B.

CONTENTS

CHAPTER 1
INTRODUCTION

This is a Reference Manual for Research Machines Extended BASIC, Versions 5 & 6, for both the network and stand-alone Computers. It is not a teaching manual and is not intended to teach you how to write good programs in the BASIC language. For this, one of the many text books on the market must be consulted. "Illustrating BASIC" by Donald Alcock, Cambridge University Press, 1977, is a concise, entertaining little book and may prove to be a good starting point. In addition "Computing Using BASIC" by Tonia Cope, Ellis Harwood, 81, is an excellent introduction to the use of BASIC on Research Machines computers.

Although this manual contains many examples, these are intended only to clarify the items described in the text, and do not necessarily illustrate how they should be used in real programs.

Information in this manual applies equally to Stand-alone disc systems and the 480Z when used as a network station except where otherwise stated.

Differences Between Versions 5 and 6 of Extended BASIC

The descriptions of those features of Extended BASIC Version 6 that are not implemented in Extended BASIC Version 5 are shown in this publication in half tone, and summarized in Appendix C.

Versions of CP/M

Both versions of BASIC described in this publication can be run on a stand-alone disc system running under the control of versions 1.4 or 2.2 of the CP/M operating system. Note, however, that the random-access file management facilities implemented in Extended BASIC Version 6 cannot be used when running under CP/M version 1.4. If you wish to use Random Access File facilities and do not have CP/M 2.2, then contact the sales office for details of upgrades. Both versions of BASIC can be used when running under CP/NOS on a network system.

How to Use this Manual

Chapter 2 is an introduction to BASIC for both network and stand-alone operation and is intended mainly for the new user. Its aim is to build up his or her confidence to the level that simple programs can be written, run, saved and reloaded from disc. In the course of this, some topics are covered that are of value to the more experienced user.

Chapters 3 to 18 form the main work of reference. The various commands and keywords are arranged roughly according to function, commands with similar functions being grouped within each chapter.

Depending on your inclination, you may prefer to read through these chapters from end to end, or merely refer to them as needed, using the contents list, index, and Quick Reference Guide.
The Quick Reference Guide is in Appendix A and is also provided as a Reference Card. Appendix B is a list of error messages, with a brief explanation of each. Appendix C lists the differences between Extended BASIC Versions 5 and 6 and previous versions. Appendix D gives instructions on setting up a printer. Appendix E is a listing of the colour lookup table used by various of the high resolution graphics routines, and Appendix F describes "Escape Sequences" which provide means of controlling from within a program certain system parameters.

Making a Working Disc

Details of making working discs for *network systems* is given the Network Manager's Guide. For *stand-alone* systems the distribution disc on which you receive BASIC from Research Machines should not be used routinely. The programs on it should be copied to a working disc and the distribution disc put away in a safe place. You may need it if your working disc becomes damaged or lost, or, if an upgrade becomes available, to return to Research Machines as proof of purchase to qualify for a reduced price for the upgrade.

The procedure for making a copy of the disc is outlined in the User Guide.

There may be several versions of BASIC on the distribution disc. To conserve space on your working copy, especially on systems with 5-inch single-density discs where you will be supplied with two distribution discs, we suggest that you erase all but the version you need and that you rename that version as BASIC.COM. Refer to the Release Note for details of the various versions.

The disc also contains a number of sample BASIC programs of varying complexity.

A Note About the Keyboard

The keyboard for your system is described in the User Guide. Some brief notes are included here to minimize any cross-referencing that maybe needed. However, from time to time the model of keyboard supplied with the 380Z or 480Z might change, so the User Guide should be consulted should the information given herein refer to a keyboard that differs from your own keyboard.

Beginners should note that the computer keyboard differs from that of a typewriter in that there are three sets of characters — the normal lower case letters, the capital letters, (obtained by holding down <SHIFT> while typing a letter), and the control characters, generated by holding down <CTRL> while pressing a letter or other character key. If you need to key in a quantity of text in upper case, a convenient way to do this without having to keep pressing <SHIFT> is to press <CAPS LOCK> once. This causes the letter keys to generate upper case characters whilst the other keys generate the unshifted characters. Thus the key marked A generates capital A whilst the key marked with numeral 1 and the exclamation mark generates 1 when pressed alone, or ! when pressed with <SHIFT> held down. Pressing <CAPS LOCK> again allows the letter keys to generate lower case characters.

Some of the older Research Machines keyboards include two keys whose markings may prove confusing. <SHIFT/3> may be marked with a pound sign but in fact it generates the hash or sharp symbol (#), used in association with file handling. On some keyboards, <SHIFT/DELT> generates the rubout code and will delete one character, but in its non-shift mode it may appear to generate a hash while in fact generating an underscore.

<RETURN> is used to signal the end of a line to the computer. <ESC> is used only within the EDIT command. Some keyboards have <ESC> marked <ALT MODE>, while on others <ALT MODE> actually generates this character. If you are unsure of your keyboard in this or any other respect, experiment.

The other special keys may or may not be connected, depending on the model of the keyboard. <FF> is equivalent to <CTRL/L> and clears the screen while <SI> is equivalent to <CTRL/O> and suppresses output. <REPT> repeats the last key pressed. <BREAK> is disabled and has no use, and <US>, <LINE FEED>, and <FS> are ignored.

The 480Z keyboard includes an additional set of special function keys on the right hand side of the keyboard. In BASIC these are mostly used for editing lines. The <CAPS LOCK> key on the 480Z keyboard stays down when pressed and is released by pressing it again.

A Note About the Display Screen

Some 380Z and 480Z systems have the ability to display text in either 40 or 80 character mode with the consequence that the number of characters per line on the screen can be changed. Where this difference may be significant is indicated within the text of the manual. Full details are given in the appropriate User Guide supplied with your system.

CHAPTER 2

GETTING STARTED WITH BASIC

This chapter is intended mainly for those meeting BASIC for the first time. It takes you on a short guided tour of some of the facilities of Research Machines Extended BASIC and is intended to be intelligible to those who have not yet learned the BASIC language. If you are such a person, we suggest that you work through this chapter step by step at the computer. You will then need to turn to one of the textbooks on BASIC programming which were mentioned in the previous chapter, using the rest of this manual for reference.

Those already familiar with BASIC but who have not yet used it on the 380Z or 480Z computer may find it helpful to scan through this chapter. It introduces many aspects of this particular dialect of BASIC. More details will be found in the later chapters.

For a network station, details of loading BASIC are given in the Network Station User's Guide. For a Stand-alone Disc System, switch on the computer or press the RESET button if it is already on. Insert a CP/M system disc in drive A and a BASIC disc in drive B, making sure that the discs are the right way up — the user guide will tell you which way this is — and that the drive doors are properly closed. Press <B> to start the operating system: the computer will display:

```
Research Machines
Release 1.1M
31K CPM vers 2.2 C

A>
```

or something very similar. Then select drive B. This is done by entering:

```
B:<RETURN>
```

in response to the "A>" prompt. Then enter the name of the BASIC interpreter followed by pressing <RETURN>. If you followed the advice in chapter 1, this will be BASIC; otherwise, check in the Release Note. When BASIC has been loaded in, it prints a message on the screen, such as:

```
RML Extended BASIC V5.0 L
Copyright (C) 1982 by Research Machines

Ready:
```

"Ready:" means that BASIC is waiting for you to enter a BASIC program line or command.

A FIRST LOOK AT THE PRINT INSTRUCTION

Enter:

```
PRINT "HELLO"<RETURN>
```

The computer will display word HELLO on the next line, like this:

```
HELLO

Ready:
```

If you make a mistake, such as misspelling the word PRINT, the computer displays the error message:

```
Syntax error

Ready:
```

If you miss out the quotation marks, the computer will display a zero:

```
0

Ready:
```

Notice that you can tell the difference between a capital letter O and the number 0 on the screen by their shapes.

What you have just entered into the computer is called a BASIC statement. PRINT is an instruction to the computer telling it to display on the screen the characters between the quotes. You can enter up to 128 characters before pressing <RETURN>. If you try to enter more characters than this, the computer will ignore them.

In the case of numbers, the number does not necessarily have to be enclosed in quotation marks. Enter the statement:

```
PRINT "99"<RETURN>
```

This instructs BASIC to print the number 99 on the next line, as expected. But if you enter:

```
PRINT 99<RETURN>
```

the computer again displays the number like this:

```
99

Ready:
```

Despite the missing quotation marks, the value is correct and there is no error message. In fact, BASIC will let you use the PRINT command for most numbers

without enclosing them in quotation marks. Here are a few for you to try:

3 −5 3.14159 1001 .303 +1.5

As you can see, a number can have a decimal point and a plus or minus sign.

USING BASIC AS A CALCULATOR

Without further study, the computer can be used as a simple desk calculator. Try entering:

```
PRINT 3+4<RETURN>
```

The answer, 7, appears on the next line. BASIC can perform five different elementary arithmetic operations:

Addition Indicated by the usual plus sign (+).

Subtraction Using the minus sign (—). The minus key on most keyboards is the one to the right of the number 0, shared with =.

Multiplication Here an asterisk (*) must be used; there is no cross symbol on the keyboard. Also, multiplication is never implied by leaving out the multiplication symbol — where in mathematics we can write "ax", in BASIC we must enter "A*X". To find 12 times 9 enter:

```
PRINT 12*9<RETURN>
```

Division Using the slash sign (/). To divide 108 by 12, enter:

```
PRINT 108/12<RETURN>
```

Exponentiation It is often necessary to multiply a number by itself a given number of times. Instead of entering:

```
PRINT 2*2*2*2*2<RETURN>
```

you can enter:

```
PRINT 2↑5<RETURN>
```

The upward pointing arrow is usually the key marked with a circumflex sign, to the right of the minus sign.

Notice that you can use a number of arithmetic operations on the same line. For example, you can enter:

```
PRINT 1+2+3+4<RETURN>
```

or

```
PRINT 1-3+6<RETURN>
```

The exact rules for combining operations will be given in Chapter 3 but you can try some experiments now if you wish.

THE IMPORTANCE OF THE RETURN KEY

So far you have been pressing <RETURN> after every line without much thought. It is worth realising why this key is used so much. The reason is simple: without <RETURN> the computer does not know when you have finished an instruction. Suppose you started entering:

```
PRINT 5+6
```

If the computer decided at this point that you had finished entering the statement and printed 11, it would not be very helpful if you had in fact planned to enter:

```
PRINT 5+6+7+8
```

which gives a completely different answer. Since the computer cannot tell when you have finished, you must tell it. You do this by pressing <RETURN>. You always have to do this after entering an instruction. Pressing <RETURN> key should be a habit by now, if you have been doing all the examples.

CORRECTING KEYING MISTAKES

No one is a perfect typist. BASIC has two features that aid in correcting keying errors.

The first is <DELT>; it acts rather like the backspace key on a typewriter and often saves you the effort of re-entering a whole line when you make a mistake. Suppose you have accidentally entered:

```
PRINT GOOD MORNIG
```

By pressing <DELT> twice you rub out the last two characters (G''):

```
PRINT GOOD MORNI
```

and you may correct the message.

Each time you press <DELT> the cursor moves one place to the left and the character it rubs out is lost.

The second special key is <CTRL/U> which is entered by holding down the key marked CTRL while pressing U. The 380Z computer responds with ↑U, and then begins a new line. The whole of the line you were entering before you entered <CTRL/U> is considered rubbed out. Thus, a second way of correcting the mistake in GOOD MORNING is:

```
PRINT "GOOD MORNIG"↑U
PRINT "GOOD MORNING"<RETURN>
GOOD MORNING

Ready:
```

Sometimes you will find that it is quicker to delete a whole line with <CTRL/ U> and start again than to rub out most of it with <DELT>.

When <CTRL/U> is entered on the 480Z, the current line is deleted from the display screen and its cursor returns to the beginning of the line. Also on a 480Z, pressing <SHIFT> key followed by the function key <F1> (on the righthand end of the keyboard) has the same effect.

GETTING STARTED WITH GRAPHICS

As well as characters, the computer can display dots. To use these, you need a way to describe what you want to display and where you want it. To specify where a dot goes, you divide the screen into columns. In 40-character mode, the screen is divided into 80 columns numbered from 0 to 79. In 80-character mode there are 160 columns numbered from 0 to 159. Column 0 is at the left and the numbers increase to the right. To simplify this introduction, it is assumed that you have a 40-character screen. The screen is also divided into 72 rows, numbered —12 to 59, starting at the bottom and increasing upwards.

Your screen is probably now full of the commands and responses you produced earlier. To clear it and make it ready for graph plotting, enter the instruction:

```
GRAPH
```

When you use this instruction, the upper twenty lines of the screen are cleared to create a "graphics area" leaving four lines for text at the bottom.

To display a dot in the middle of the graphics area (column 40, row 30), you enter:

```
PLOT 40,30,2
```

The digit 2 indicates to BASIC that you want to display a white dot. You can change it to a grey dot, enter:

```
PLOT 40,30,1
```

To rub out a dot at column 40, row 30, enter:

```
PLOT 40,30,0
```

To put a dot in the lower right corner, you must specify column 79, row 0:

```
PLOT 79,0,2
```

Notice that you specify the column first, as you do the X coordinate in graph plotting. Now try missing out the third number:

```
PLOT 0,59
```

The brightness you last specified (white) has been remembered; a white dot is plotted in the top left corner of the screen.

You have probably noticed that the last few instructions that you have entered have moved up the screen for four lines and then disappeared. To put things back as they were before you entered the GRAPH instruction, enter the instruction:

```
TEXT
```

This instruction moves the lines you enter up to occupy the whole screen.

A SIMPLE PROGRAM

So far, the PRINT and PLOT commands have been acted on immediately, as soon as <RETURN> was pressed. This way of using BASIC is called direct mode or immediate mode. It is also possible to hold a series of commands in the computer's memory and then execute them later. In this mode, the sequence of stored statements forms a BASIC program. The program can be executed once or many times, and can be later stored on a disc to be reloaded and executed at some time in the future. Clearly, program mode is far more useful than direct mode.

To enter a number of lines of commands as part of a program, you must start each of them with a line number. To make sure the computer's memory is cleared, enter:

```
NEW<RETURN>
```

and then enter the statement:

```
10 PRINT 99<RETURN>
```

This time nothing happens except that the cursor moves to the beginning of the next line; the number 99 is not printed on the screen. Now enter RUN (followed by <RETURN>):

```
RUN<RETURN>
99

Ready:
```

The RUN command causes the program currently being "looked at" by BASIC to be executed. To see the program on the screen, use the LIST command:

```
LIST<RETURN>

10 PRINT 99

Ready:
```

NEW, RUN and LIST are called utility commands. They are described in Chapter 4.

Now try a more ambitious program:

```
10 LET X=0
20 LET X=X+1
30 PRINT X
40 GOTO 20
```

If you make a mistake while entering these lines, use <DELT> or <CTRL/U> as described previously. When you have entered in the four statements, use LIST to make sure they are all correct. If you have made a mistake in one of them, you need only re-enter the whole line. BASIC holds the most recent version of a line with a given line number. Notice that you did not need to enter NEW before keying in this example as line 10 of the previous example was automatically replaced with the new line 10.

Having checked that the program is correct, enter RUN. The program should print a sequence of numbers, starting from 1, the numbers being the current values taken by the variable X. A variable is the term used to refer to a place in the computer memory where a number is stored. In BASIC, variables are given names of one or two letters; they are described in more detail in Chapters 3 and 6.

When you run this program, it is probable that it will pause after displaying the number 22 with the cursor blinking on the bottom line. If not, press <CTRL/A>.

CONTROLLING SCREEN OUTPUT

We assume that you have run the previous example and that the cursor is blinking at the bottom of the screen. This demonstrates the auto-paging function of the computer.

After the LIST command has been used for a long program, output to the screen pauses as soon as the screen has been filled. To see the next lines in the program press the space bar. Try this a few times.

Auto-paging is turned on and off by <CTRL/A>, which acts like a switch. Press <CTRL/A> once and observe that BASIC no longer pauses at the end of each page. Press <CTRL/A> again to resume auto-paging.

There is another way of stopping output to the screen which is useful when you want to stop output at a specific point. Turn off auto-paging, then press <CTRL/S> to stop output. Note that pressing the space bar no longer causes output to resume; instead you must press <CTRL/Q>. <CTRL/S> and <CTRL/Q> do not alter the auto-paging switch.

INTERRUPTING A PROGRAM

It will probably be evident by now that if we did nothing, the example program would continue printing a series of increasing numbers for ever, or very nearly so. There is a need for a method of interrupting a running program without having to press the RESET button. Press <CTRL/Z> to achieve this. Try it, and observe that BASIC displays:

```
Interrupted at line n

Ready:
```

where "n" is the number of the line that was about to be executed when <CTRL/Z> was pressed. The keyboard is checked for <CTRL/Z> at the beginning of every statement. This means that a BASIC program can be interrupted at any time.

If necessary, an interrupted program can be resumed by entering:

```
CONT
```

This continues execution from the point of interruption. Try this, and then interrupt the program again with <CTRL/Z>.

PROGRAM MODIFICATION

The program example is clearly rather unmanageable. We would rather it stopped after printing a defined number of values of X. To make sure that the program is interrupted, enter the additional statement:

```
25 IF X>15 THEN STOP
```

Although we have not yet met the IF command, the meaning of this statement should

be fairly clear. This addition illustrates a number of points.

Enter LIST and observe that although line 25 was the most recently typed, BASIC has inserted it in the correct place, between lines 20 and 30.

Gaps of 10 were left in the original numbering of the program lines, just in case we needed to add an additional line. It is common practice to number BASIC statements in increments of 10. The numbering may be tidied up at any time by the RENUMBER command. See Chapter 4 for more details.

We have now modified the program by adding an additional statement. It can no longer be continued by the CONT command.

Try out the modified program and make sure that it stops after the number 15 has been displayed.

USING A PRINTER

You should skip this section if you do not have a printer attached to your computer.

Once a program is working correctly, it is usually desirable to make a permanent record of it and of its output on a printer. If using BASIC on a stand-alone system it is first necessary to select a printer option. The computer supports various types of printer and the user must inform the computer which type is attached. If no printer option is selected, output intended for the printer is sent to the screen.

A number of ways of selecting a printer option are available. Connecting and setting up a printer on a new system can be difficult, particularly if the printer was not purchased through Research Machines. Beginners are therefore advised to leave this until they are more familiar with the rest of the system. If the equipment is already operational, refer to chapter 4 and appendix D for setting printer options.

Once a printer option has been selected, the LLIST command can be used. LLIST is similar to LIST but its output goes to the printer rather than the screen. Similarly, PRINT statements can be replaced by LPRINT, with similar effect.

An alternative and often more convenient method is to press <CTRL/P>. After <CTRL/P>, all output to the screen is also sent to the printer, until <CTRL/P> is next entered. Thus, a program can be tested with its output sent to the screen and then run again without modification, merely by entering <CTRL/P> to produce a fair copy on the printer. <CTRL/E> is equivalent to <CTRL/P>.

If you have a printer connected to your system, test these new commands on the modified program.

If you are using a network system your output will be sent to a file on the network

server, and printed only after you indicate that a suitable quantity has been produced. You do this by issuing the command CLOSE #2. See Network Users Guide for further details.

SAVING AND LOADING PROGRAMS ON DISC

Now that you have a working program, you may want to save it on disc so that it can be run on another occasion without the effort of keying it in again. The SAVE command is used. Assuming that you would like to call the program TEST1, enter:

```
SAVE "TEST1"<RETURN>

Ready:
```

On a stand-alone disc system the light on one of the disc units will come on for a few seconds before "Ready:" is displayed. On a network system the light on one of the disc units attached to the Network Server will come on for a few seconds before "Ready:" is displayed. You can confirm that the program has been saved by using the DIR command:

```
DIR<RETURN>
B: TEST1    BAS

Ready:
```

The letter before the colon is the logical disc unit containing the disc on which the program has been saved. BAS is the extension of the filename (secondary filename) which is given automatically by BASIC if none is supplied.

Saved programs are reloaded by the LOAD command. Before trying LOAD, delete the existing program from memory by using the NEW command and confirm that you have done so by using the LIST command. It is not necessary to do this as BASIC automatically deletes existing programs from memory before loading a program, but doing so at this stage will help to confirm that you can save and reload programs correctly. Having cleared memory, reload TEST1 by entering:

```
LOAD "TEST1"<RETURN>

Ready:
```

Again the disc unit light will come on and there will be a short pause before BASIC displays "Ready:". You can now confirm that TEST1 has been reloaded by using the LIST or RUN commands.

ENDING A SESSION

At the end of a BASIC session, after any new programs have been safely saved on disc, you will want to leave BASIC. In fact all you need do on a stand-alone system is to take your discs out of the disc drives and turn the computer off. However, you may wish to return to the CP/M Operating System (CP/NOS for the network user) to use one of the other system programs. In this case you either enter <CTRL/C> or the BYE command. Because <CTRL/C> can be used accidentally, the warning message:

```
↑C--Are you sure (Y/N):
```

is displayed. Enter Y or y to leave BASIC or N to continue using BASIC.

SOME COMMON PROBLEMS

This section refers to stand-alone disc systems. Two problems can arise whilst saving programs in Extended BASIC, neither of which is dealt with in a particularly elegant manner at present.

On a stand-alone disc system you may decide that you wish to save the current BASIC program on a different disc. You change the disc, then issue the SAVE command. Instead of saving the program, the system displays an error message such as:

```
BDOS ERROR ON A: R/O
```

If you press any key except <CTRL/F>, you might lose your program. Refer to the descriptions of the RESET and SAVE commands in Chapter 4 for the recovery procedure. Remember always to issue the RESET command after changing a disc.

Alternatively, all is going well during a SAVE operation until a message such as:

```
A 00 15 0A 31F0 BDOS ERROR ON A: BAD SECTOR
```

appears. This sort of message means that there is an unreadable sector on the disc. Probably the safest thing to do is to remove that disc for later investigation, replace it with another disc and enter <CTRL/F>, followed by J>103. This restarts BASIC at its recovery address. Your program should be intact. Now issue the RESET command (because you have changed discs) and try the SAVE command again.

If problems arise when saving Extended BASIC programs on the network, refer to Appendix B, and if the error is not described there, refer to the Network Station Users Guide for details of system error messages.

CHAPTER 3

ELEMENTS OF BASIC

This chapter discusses some of the elements from which a BASIC program is constructed. Some knowledge of the BASIC language is assumed, so it is probably best to skim through this chapter on the first reading and then study it more carefully after reading some of the later chapters in this manual.

FORMAT OF NUMBERS

When a number is supplied to BASIC from the keyboard, either as a constant that forms part of a program or in response to an INPUT instruction, it may take a variety of forms. For example:

3 −5 3.14159 1001 .303 +1.5

As can be seen, a number may consist of an integer part (before the decimal point) and a fractional part (after the decimal point), either of which may be omitted, and may be preceded by a plus or minus sign. If no sign is supplied, the number is assumed to be positive.

BASIC displays or prints numbers in a similar form. If the number is positive, it is preceded by a space, and by a minus sign if it is negative. Numbers larger than 999999 or smaller than 0.01 are printed in scientific (or "exponential") notation. For example, the number 1234567 is printed:

```
 1.23457E+06
```

This number should be read as "1.23457 times 10 to the power 6". Similarly, the value 0.0001 is printed:

```
 1E-04
```

meaning "1.0 times 10 to the power minus 4".

Numbers may be input to BASIC in normal or scientific notation. In scientific notation, lower case "e" is also recognized as indicating an exponent. The change in format for large or small numbers applies only on output.

In extended BASIC version 6, the format of numbers can be changed by use of the USING command (see chapter 14).

Numbers are stored to a precision of approximately 6 significant digits. The range of values allowed is approximately −1E38 to 1E38, with values whose magnitudes are

less than approximately 1E—38 being indistinguishable from zero.

Checks are made for meaningless exponents in numerical constants. Very large positive exponents cause arithmetic overflow while large negative exponents generate zero.

A number may also be supplied in hexadecimal, in which case it is preceded by an ampersand (&) and consists of a sequence of hexadecimal digits (0—9, A—F). For example:

&1A &0D &FC00

The value of the constant must lie within the range 0 to &FFFF.

LITERAL STRINGS

A "literal string" is a sequence of characters enclosed in quotation marks ("). The characters must not include a quotation mark but may include any other printable character. In a DATA statement, the quotation marks are optional if the string does not contain commas, colons, or leading spaces.

NUMERIC VARIABLES

"Variable" is the term used to refer to a named entity whose value can change during execution of a program. A numeric variable holds a numeric value that may change during execution of a program. A variable name must begin with an alphabetic character (A to Z). This allows the computer to distinguish it from numeric constants (which always start with a number). After the first letter, the variable name may consist of any alphanumeric character (A to Z, 0 to 9). Examples of variable names are:

B B3 ZP X Y5

To improve program readability, you can use variable names that are meaningful, such as:

SUM MEAN AVERAGE

Only the first two characters are used to distinguish one variable name from another. Thus the names COUNT and COCOA refer to the same variable.

CAUTION: variable names must not contain embedded BASIC instruction names. For example, the variable name LONG would give rise to a syntax error since it contains the BASIC instruction ON. Furthermore, since Extended BASIC Version 6 supports more instructions than Version 5 (such as MIN, MAX, TYP), it is possible that programs that run under Version 5 will fail under Version 6 unless certain variables are renamed.

STRING VARIABLES

All the variable names described in the previous section are used to hold numeric values that are to be used in calculations. A variable may also hold a string of alphabetic, numeric, or special characters, such as "HELLO" " 100,57", or "25%"; these variables are called string variables. A string variable name has a dollar sign ($) appended. For example:

```
A$      B9$      MESSAGE$
```

Note that the last example would be indistinguishable from ME$. A string variable may hold from 1 to 255 characters, or it may be empty. An empty string variable is said to contain the null string. It is not possible to use string variables in arithemitic operations to calculate new values.

ARRAY VARIABLES

The numeric and string variables described above are referred to as simple variables, since each can hold only one numeric value or one string of characters. A numeric or string array can hold a series of numeric values or strings. An array variable is a member of such an array and has the form of a simple variable name followed by a list of subscripts enclosed by parentheses. Each subscript is a number or an arithmetic expression that evaluates to a number in the range 0 to 65535. There can be any number of subscripts, up to a limit of 255. Smaller limits will in practice be imposed by the amount of available memory. Examples of array variables are:

```
B(2)      B3$(4)      SUM(5,3)      MESSAGE$(10)
```

If there is a single subscript, as in the first example, the array variable can be thought of as a member of a single row of simple variables. Thus B(2) is a member of the array B(0), B(1)...B(n). Similarly, array elements with more than one subscript form part of a matrix containing two or more rows of variables.

The number of rows in an array and the number of elements in each should be established by a DIM statement before referring to one of its elements (see Chapter 6). Thus the statement:

```
DIM B(20)
```

defines numeric array B to have one row containing 21 elements numbered 0 to 20. If an element of an array is referred to before the array has been defined by a DIM statement, BASIC creates it automatically with as many dimensions as there are subscripts and with eleven elements in each dimension (allowing subscripts in the range 0 to 10).

If any subscript value exceeds the number of elements defined for its row, the error message:

```
Subscript out of range
```

results. A simple variable can have the same name as a subscripted variable without confusion (by the BASIC interpreter, at least).

Note that a subscripted string variable represents an element of a string array, and is itself a complete string. A simple string must not be defined with a DIM statement to reserve storage for the characters in it, as is the case with many other BASICs.

STORAGE REQUIREMENTS

A simple numeric variable occupies six bytes of storage, the first two holding the name and the last four holding the internal representation of the number (see below). A string variable also occupies six bytes, again using two bytes for the name and four for the internal representation of the string. In addition to these six bytes, one byte of memory is required for each character in the string. These bytes may be located in the program itself, for example after statements such as:

```
100 LET A$="HI THERE"
```

or in "string space", described below.

A numeric array element occupies just the four bytes necessary to hold the internal representation of the number. The entire array occupies five bytes plus two bytes per dimension plus the four bytes per element. String arrays require the same amount of storage as numeric arrays, plus space to store the actual strings.

String space is an area at the top of memory which is used for storing the characters held in strings. The size of string space is fixed by the CLEAR command (see Chapter 6), or 100 bytes by default. Where possible, literal strings occurring within the program are used in place, to reduce storage requirements. When string space becomes exhausted, a "garbage collector" is used to recover the space released when strings are reassigned. This process can cause an observable delay of up to a few seconds or so. If there is still insufficient space the error message:

```
No string space
```

is produced.

USE OF MEMORY

Extended BASIC Version 5 uses the first 13K bytes or so of memory for the interpreter itself and some workspace. The Extended BASIC Version 6 interpreter is about 20K long. Above this come in turn the program, the simple variables, and the arrays. Next is a gap (of a size determinable by the function FRE) which is unused. Above the empty space is the stack, which is used to store return addresses for GOSUB statements and information about FOR loops, etc. Above the stack is the string space, the area used for storing the characters held in strings. At the top of memory is "cache", an area that BASIC does not use at all and that is set to a size of zero bytes when BASIC is started. Cache memory is intended for use with machine language routines, especially the routines GLOAD and GSAVE used in the High Resolution Graphics package. See also Chapter 13.

ASSIGNMENT OF VALUES

During the execution of a BASIC program, values can be stored in either numeric or string variables by means of the LET instruction. Examples are:

```
LET A=2
LET M$="HALLO"
LET B(5)=3.14159
```

This process is referred to as assigning a value to a variable. Many different values may be assigned to a variable during the course of a program. The LET command is described in Chapter 6.

Extended BASIC automatically assigns the value zero to numeric variables and a null string value to string variables on their first occurrence in a program. Thus, if variable C has not yet been used in a program, the statement:

```
LET A=C
```

assigns zero to A. However, it is a dangerous programming practice to assume that this will occur, for not all dialects of BASIC used on other computer systems work the same way. You are urged to assign values to all variables before using them. Note that a statement of the form:

```
LET M$=""
```

assigns a null string to a string variable.

Extended BASIC Version 6 assigns non-zero values by default to certain variables. See chapter 9.

EXPRESSIONS

In addition to consisting of a numeric or string constant, the right hand side of a LET statement, and indeed most other places where such a constant can occur, can consist of an expression. An expression is composed of operators and operands, which must alternate and end with an operand. An operand is either a constant, a variable, a function call, or an expression enclosed within parentheses. An operator is an arithmetic operator, a relational operator, a logical operator, or the string concatenation operator.

Examples of arithmetic operators are

```
LET A=B+2       addition
LET C=D-5       subtraction
LET P=5*1.5     multiplication
LET X=100/3     division
LET E=2↑3       exponentiation
```

Relational operators are usually used in conditional statements of the form:

```
IF A>3 THEN PRINT "A GREATER THAN 3"
```

The IF instruction and the available relational operators are described in chapter 8. Relational operators can be applied to either numeric or string operands, but the operands must be of the same type. The relationship between two strings is decided by the ASCII value of the first pair of differing characters starting from the left. Thus "APPLE" is considered to have an ASCII value that is less than that of "AXIOM".

The logical operators AND, OR, and NOT are normally used to decide the truth or otherwise of a series of relationships, such as in the statement:

```
IF A>3 AND B=2 THEN ...
```

where the AND operator combines the results of the two relational operators. However, the mechanism used is to combine the values of the operands, expressed as 16-bit binary values, with a non-zero result being regarded as "true" and a zero result as "false". Thus you can write statements, such as:

```
IF A AND 1 THEN PRINT "A IS ODD"
```

The effect of the AND operator is to perform a logical AND operation on the corresponding bits of its operands, in this case A and 1. The result of this operation is either 0 or 1 depending on whether A is even or odd. 0 is false, and 1 is treated as true.

Relational operators can be used as part of non-relational expressions. A true result is returned as —1, and false as zero. For example, the following statement adds one to the variable B if A is negative:

```
LET B=B-(A<0)
```

The string concatenation operator is the plus sign (+). It is used to join one string to another.

For example, the statement:

```
PRINT "ROSE" + " " + "BUSH"
```

joins three strings to print:

```
ROSE BUSH
```

The order of priority of the operators is:

+	String concatenation
↑	Exponentiation
-	Unary minus
* /	Multiplication Division
MOD	Modulo arithmetic
+ -	Addition Subtraction
MIN	Minimum Value
MAX	Maximum Value
< <= > >= = <>	Relations
NOT	Logical NOT
AND	Logical AND
OR	Logical OR
XOR	Logical XOR

with the highest priority operators in an expression evaluated first. Operators of equal precedence are evaluated from left to right as given in an expression.

These rules of operator priority may be modified when necessary by parentheses. Expressions in parentheses are evaluated first. Thus:

```
LET A=3+4*5
```

assigns 23 to A, while:

```
LET B=(3+4)*5
```

assigns 35 to B.

For the description of functions MIN, MAX, MOD and XOR see Chapter 9.

FORM OF A BASIC PROGRAM

A BASIC program is made up of lines, each of which consists of a line number and one or more statements. The line number is a number between 1 and 65529. Note that line 0 is not permitted. It is normal to have one statement on each line. However, when space

is short it is permissible to put more than one statement on a line, the statements being separated by colons (:) as in line 20 below:

```
10 LET A=3
20 LET B=4 :LET C=5
```

There can be as many statements on a line as will fit, the length of a line being limited to 128 characters. Spaces may be included freely to improve readability but in this version of BASIC their use is not required. The last example could be keyed in as:

```
20LETB=4:LETC=5
```

but is much more difficult to read than when spaces are used. Spaces are not allowed within the names of BASIC keywords.

BASIC statement lines can be added to a program in in any order. BASIC rearranges them internally into ascending line number order. When a program is executed, the statements are executed in ascending line number order unless control is transferred to another line number by a statement such as GOTO. Only the most recent version of a line with a particular line number is retained. Entering a line number of a line already present in a program followed immediately by <RETURN> causes that line to be deleted, including its number.

INTERNAL REPRESENTATION OF NUMBERS

Numeric constants and the values of numeric variables are input and output by BASIC in decimal form and are held internally in a special binary floating-point (exponential) format. In Extended BASIC, 24 bits are assigned to the binary fraction and its sign, and 8 bits to the sign and magnitude of the binary exponent. This form is similar to the "scientific" format in which BASIC prints large and small numbers, except that the fraction always lies between a half and one and is raised to a power of two rather than ten. Futher details are given in the description of the VARADR command in chapter 18.

The format allows a precision of somewhat better than 6 significant decimal places; when converted back to the decimal system, numbers are output to 6 significant figures with the last figure rounded. Thus the fraction 2/3 (two-thirds) is represented as 0.666667. As mentioned above, the range of numbers that can be represented is approximately 1E+38 to 1E—38. Numbers less than the lower limit are rounded down to zero whilst numbers greater than the upper limit result in the error message:

```
Arithmetic overflow
```

and cause program execution to stop.

This form of representation can sometimes give unexpected results. Although such results are not usually a problem in day to day work, the programmer should be aware

of the possibility of an apparently erroneous result occurring. The following examples illustrate what can happen.

Some numbers, although they can be represented exactly in the decimal system, cannot be held exactly in the computer. The fraction 1/10 is an example. In decimal this is 0.1 and you know that if you multiply it by 1000, the result will be 100. This is also true in the computer. The program:

```
10 LET T=1/10
20 PRINT T*1000
```

prints 100. You also know that if you were to add 0.1 to itself a thousand times you would still get the result 100. But this is not so in BASIC. The program:

```
10 LET T=1/10
20 LET S=0
30 FOR I=1 TO 1000
40 LET S=S+T
50 NEXT
60 PRINT S
```

prints 99.9991 instead of 100. What has gone wrong? The answer lies in the fact that the fraction 1/10 cannot be held exactly. When we multiplied it by 1000, BASIC's automatic rounding mechanism took care of the slight residual error and gave the correct result. But when you added in the slightly incorrect value of T a thousand times, you compounded the error a thousand fold; automatic rounding was insufficient to compensate.

In a program you might need all the intermediate values of S. If so, a solution is to recalculate S each time through the loop, so that each intermediate value of S contains only one error. For example:

```
10 FOR I=1 TO 1000
20 LET S=I*0.1
30 NEXT I
40 PRINT S
```

The PRINT statement in line 40 shows that the final value of S is now correct.

The problem can be summed up as an error due to the accumulation of many smaller errors. It can occur in the majority of implementations of computer languages, particularly on mini and microcomputers.

The second example is less obvious. Several of the operations in BASIC convert numbers from their internal binary floating-point form to an exact whole number before use, a particular example being the evaluation of array subscripts. In Extended BASIC, as in the majority of dialects of BASIC, this is done by truncation rather than by rounding. Thus A(2.5) refers to A(2), not A(3). The situation is made more confusing by the fact, already mentioned, that the PRINT instruction carries out slight

rounding. It is thus possible for a variable to have one value when printed and another when used as an array subscript, the second value being one less than the first, as the following example shows:

```
10 LET T=1/10
20 LET S=0
30 FOR I=1 TO 30
40 LET S=S+T
50 NEXT I
60 PRINT S              When printed, S is 3
70 DIM A(3)
80 LET A(2)=2
90 LET A(3)=3
100 PRINT A(S)          But this prints 2!
```

The internal value of S is in fact fractionally less than 3; when printed it is rounded up to 3, but when truncated to a whole number yields 2.

Fortunately the problem is rare, tending to happen only with very large numbers and with smaller numbers when there have been many intermediate calculations, as in this case. A simple solution is to force rounding rather than truncation to take place by adding 0.1 to the subscript. Changing line 100 to:

```
100 PRINT A(S+0.1)
```

prints 3, as expected.

Users may be reassured to know that Extended BASIC follows Dartmouth BASIC in using truncation to evaluate array subscripts. The problem is not confined to the 380Z and 480Z computers.

CHAPTER 4

GENERAL PURPOSE UTILITY COMMANDS

In response to the "Ready:" prompt, the user can enter a command either with a line number (program mode) or without a line number (immediate or direct mode). In program mode, the user enters a sequence of BASIC command lines, each prefixed with a unique line number, and when all requisite lines have been entered, the program can be run (executed) by entering RUN<RETURN>. In immediate mode, a command is entered without a line number prefix and is executed as soon as <RETURN> is pressed.

Most of the commands in this section are normally used without line numbers, in direct mode. However, some commands, ERASE and RENAME for example, are often used within programs.

Commands which are only available with Extended BASIC Version 6 are shown in half tone.

Several of the commands described in this section require a file specification, which may be supplied within quotation marks, or in a string variable or expression. Examples are:

```
LOAD "CIRCLES"
ERASE F$
```

The file specification comprises a filename of up to 8 characters drawn from the character sets A-Z, a-z, 0-9 and $. Lower case letters are converted to the corresponding upper case letters. The filename may optionally be preceded by the name of the disc drive containing the disc onto which the file is to be written followed by a colon, as in the following example:

```
SAVE "B:GRAPH"
```

If the disc drive name is omitted, the drive used when BASIC was loaded is assumed. A three-character extension to the filename (secondary filename) may be supplied, preceded by a full stop. If omitted, .BAS is assumed.

In addition, the output of commands such as LIST and DIR can be redirected to the printer or to a file. The mechanism for doing this is described in Chapter 12.

AUTO

```
AUTO      AUTO 5,15      AUTO ,20      AUTO 100
```

Each line of a BASIC program must be preceded by a line number. The AUTO command provides a method of generating line numbers automatically. Once an AUTO command has been entered, a new line number appears on the screen each time <RETURN> is pressed. The

sequence of line numbers required may be defined by the user. If the user enters AUTO without a numeric value, the first line number to appear will be 10 and the value of each subsequent line number generated will be incremented by 10. The user may choose an alternative initial line number and/or incremental value by specifying the required values within the AUTO command. The command AUTO 5,15 will generate the sequence of lines numbers: 5 20 35 50 .... The command AUTO ,20 generates the sequence 10 30 50 70 ....

An incremental value of 10 is assumed when the AUTO command does not specify an incremental value. Thus the command AUTO 100 generates the sequence 100 110 120 130 ....

To warn the user that a line number generated by the AUTO command is the same as a line number that currently exists in the program, a + (plus sign) is prefixed to it. If the user does not wish to overwrite the existing line, it is necessary to press <RETURN> to generate the next line number in the sequence without deleting the existing line.

Any line number in the range 1 to 65529 is valid. Exceeding this range causes the error message "Arithmetic Overflow" to be displayed.

The AUTO command is terminated by pressing <CTRL/Z>, which also has the effect of cancelling the line currently being entered. The AUTO command must be terminated, for example, before entering the RUN command in order to execute the program.

The "." form of line number

In direct mode commands, a period (".") can be substituted anywhere that a line number is normally used.

The period represents the last line accessed by a program editing or execution command. For example, to correct an erroneously-entered line that has just been entered, use the EDIT command with the period:

```
10 A$=MID$ (B$,5,6
EDIT.
10
```

In response to the immediate command EDIT. the line most recently referred to is copied into the edit buffer ready for correction by the user, as described in chapter 5.

After an execution error, the period is set to the number of the line on which the error occurred, so that the erroneous line may be listed by the command:

```
LIST.
```

Note that an attempt to substitute "." for a line number in program mode produces the error message "Undefined Statement".

CONT

```
CONT          CONTINUE
```

After pressing <CTRL/Z>, or after a program has executed a STOP or END instruction, the message:

```
Interrupted
```

is displayed. You may resume execution by entering CONT. While the program is stopped, PRINT, LVAR or LET may be used to display or change the values of variables. CONT cannot be used after program modification or a runtime error.

DELETE

```
DELETE 25        DELETE 25-75
DELETE 25-       DELETE -75
```

These commands cause the deletion of lines 25, 25 to 75, 25 to the end, and lines up to line 75 respectively.

COPY

```
10 COPY 170,15=40-60
```

Any group of program lines from one part of a BASIC program can be copied to another specified part of the program by means of the COPY command.

The form of the COPY command is:

COPY n1[,n2] = n3—n4

The right hand side of the "=" sign specifies the range of line numbers to be copied (i.e. n3 and n4 are the first and last line numbers of the group of lines to be copied).

The left hand side of the "=" sign indicates where the copied lines are to be placed in the program and, optionally, the increment to be used for the line numbers of the copied lines. If no increment is specified, an increment of 10 is assumed by the COPY command.

The group of program lines are copied in the same format as produced by the LIST command.

Before lines are copied the new line numbers are validated to ensure that they do not fall within the range of line numbers being copied. If such overlapping of line number values occurs, the error message "Illegal Function" is displayed.

Any references to other line numbers displayed within the block of lines to be copied (e.g. within GOTO statements) are not renumbered by the COPY command.

Example:

```
10 I=I+1
20 IF I=10 THEN 50
30 PRINT I
40 GOTO 10
50 END
```

To copy lines 10 to 40 of the original program to line 200 onwards of a new program (and to use a line increment of 5 for the copied program lines) the following command is used:

```
COPY 200,5=10-40
```

New Program:

```
 10 I=I+1
 20 IF I=10 THEN 50
 30 PRINT I
 40 GOTO 10
 50 END
200 I=I+1
205 IF I=10 THEN 50
210 PRINT I
215 GOTO 10
```

Note that the line numbers within the copied statements in this example (50 in copied statement 205 and 10 in copied statement 215) have *not* been renumbered by the COPY command.

DIR

```
DIR          DIR "B:*.*"
```

This command lists a disc directory on the screen in CP/M format. The first form lists all files on the logged on disc which have an extension of .BAS. The second form can be used to perform other directory functions, the example listing all files on unit B. Note that on a Network, or when running under CP/M 2.2, files which have been given the $SYS attribute are not displayed.

ERASE

```
ERASE "STATS3"
```

The specified file is erased. No error is reported if the file does not exist. The filename must be unambiguous, i.e. it cannot contain * or ?.

You should not erase a file that is open, as this can cause odd results, particularly on a Network System. The command CLOSE INPUT can be used to close input files. The command CLOSE closes output files (See chapter 12).

FSAVE

```
FSAVE "CIRCLES"
```

FSAVE acts like SAVE (described below), except that the program is saved in internal format. The advantage of internal format is that loading (see below) is very much faster, by a factor of five or more on large programs. However, programs saved using FSAVE are very much less portable than programs saved using SAVE and an internal format program saved using one version of BASIC cannot be loaded by another version of BASIC. The chances of recovering from a corrupted file are also very much less with internal format. Internal format files cannot be combined by the MERGE command.

Because of these restrictions, it is suggested that FSAVE be used sparingly, and that you should never leave yourself in the situation where your only copies of a program are in internal format. (Recreating a BASIC program in ASCII format from its internal format version can sometimes be done using LOAD and SAVE, but the success of this depends upon the version of BASIC being used to read the internal format file being identical to that used in the original FSAVE.)

LIST

```
LIST    LIST 25    LIST 25-75
LIST 25-    LIST -75
```

The specified lines of the program are to be displayed on the screen. The various forms display the whole program, line 25, lines 25 to 75, from line 25 to the end, and from the beginning to line 75.

The display may be interrupted by pressing <CTRL/Z>, or stopped temporarily by pressing <CTRL/S> and resumed by pressing <CTRL/Q>. Automatic paging is turned on and off by <CTRL/A>. When it is turned on, the display stops after the screen has been filled with lines of program. The cursor blinks to indicate that further lines can be displayed; press the space bar to display the following lines.

LLIST

```
LLIST    LLIST 25    LLIST 25-75
LLIST 25-    LLIST -75
```

Similar to LIST but the program is listed on the printer. A printer is normally selected by the PRINTER command (see below). If no printer option has been selected, the listing is displayed on the screen.

On a network system the output will be sent to the network printer unless a local printer has been selected (see below). You must terminate this listing with CLOSE #2 (See chapter 12).

LOAD

```
LOAD "STATS1"     LOAD "B:STATS1.BAS"
```

The LOAD command loads a program from disc storage. This

command must include the name of a program, either enclosed in quotation marks or as a string variable or expression. This enables the disc directory to be searched for the desired program. If the program, which can be in either internal format or the normal ASCII format, is found, it will be loaded and BASIC will display the prompt "Ready:". Any program previously being entered or run is lost unless saved.

Not finding the program on the disc results in the message:

```
File not found
```

and finding the program in the wrong internal format gives:

```
Wrong internal format
```

LOAD? `LOAD? "CIRCLES"`

LOAD? will verify that a file on disc is the same as the program held in memory. It will only work on internal format files, and an attempt to use the LOAD? command to load an ASCII file will generate the message:

```
Can't verify ASCII files
```

Any difference between the files on disc and in memory results in the message:

```
Files different
```

otherwise BASIC merely displays the prompt:

```
Ready:
```

In neither case will either program be altered in any way.

LOADGO `LOADGO "CIRCLES"` `LOADGO "PROG",100`

LOADGO loads the specified file, in either internal or ASCII format, and starts executing it. If no line number follows the filename, execution of the new program is started at the lowest numbered line, otherwise at the specified line. If no filename is given, the error message:

```
Missing file name
```

is produced. LOADGO is also available as a CP/M command line. Thus, from CP/M, the command:

```
BASIC CIRCLES
```

would load and run the program CIRCLES. In this case, no line number

line number is permitted. If the program is in the wrong internal format, the message:

```
Wrong internal format
```

is displayed. Note that the file name must not be enclosed in quotation marks when given from the CP/M command line.

Like all commands which modify a program, LOADGO initializes all variables and user-defined functions before executing it.

MERGE `MERGE "STATS2"`

The MERGE command allows you to combine lines of a program from a file on disc with a program already in memory. The command has a form similar to the LOAD command, in that it is followed by the name of the program to merge, enclosed in quotation marks or held in a string variable or expression. MERGE is similar to LOAD except that a pre-existing program line is only replaced when an incoming line has the same number. Otherwise the new lines are added. The RENUMBER command is useful for changing existing line numbers in order to prevent lines being inadvertently deleted when merging in additional ones with the same line numbers.

The file must not have been previously saved using the FSAVE command. Attempting such a merge results in the message:

```
Can't MERGE internal files
```

MERGEGO `MERGEGO "FUNX"` `MERGEGO "CIRCLES",100`

The MERGEGO command is a combination of the LOADGO and MERGE commands described above. The specified file is merged into the current program and the new combined program is executed, starting at the specified line number if one is given, otherwise at the lowest line number. As with MERGE, MERGEGO cannot cope with an internal format file, and if presented with one produces the error message:

```
Can't MERGE internal files
```

Also as with MERGE, any lines in the memory resident program with the same line numbers as lines in the program to be merged into it are overwritten.

As with all commands that modify the program, MERGEGO initializes all variables and user-defined functions before executing it.

The following program exemplifies the use of MERGEGO:

```
 10 INPUT "Function"; A$
 20 ERASE "TEMP"
 30 CREATE #10, "TEMP"
 40 PRINT #10, "70 DEF FNY(X)="; A$
 50 CLOSE #10
 60 MERGEGO "TEMP", 70
 80 GRAPH : TEXT
 90 FOR X=0 TO 79
100 PLOT X, FNY(X), 2
110 NEXT X
120 INPUT "Another"; A$
130 IF A$="Y" OR A$="Y" THEN 10
```

These commands are explained in Chapters 6, 7, 9, 11, and 12.

NEW `NEW`

The program, if any, is deleted. All variables are lost. The command is used to clear the program area prior to keying in a new program.

PRINTER `PRINTER 3` `PRINTER 4,6`
LOCAL PRINTER `LOCAL PRINTER 2,4` `LOCAL PRINTER`
NETWORK PRINTER `NETWORK PRINTER`

The PRINTER command allows you to select a printer option directly from BASIC, without entering the Front Panel (see the System manual). It also allows printer selection under program control. The first argument selects which printer type to use, and the second, if present, is the baud code for serial devices. If a baud code is supplied for an option which does not require it (e.g. the parallel interface) then it is ignored. If the baud code is omitted for an interface which requires one (e.g. an SIO-4) then baud code 0 will be assumed. See Appendix D or the System manual for details of the significance of the terms "printer type" and "baud code".

On a network system there exists the possibility of connecting a local printer to a station as well as a central printer serving the whole network.

PRINTER sets options for a local printer only.

LOCAL PRINTER directs all printed output to the local printer instead of the network printer, and may be followed by "printer type" and "baud code" parameters as in the PRINTER command.

NETWORK PRINTER directs all printed output to the network printer instead of the local printer, and may *not* be followed by printer options. To set network printer options, use the program called CONFIG on your Network Maintenance disc, as described in the Network Users Guide.

RENAME

```
RENAME "NEWFILE", "OLDFILE"
```

The file OLDFILE.BAS is renamed to NEWFILE.BAS. The error message "File not found" is reported if OLDFILE.BAS is not found. If NEWFILE.BAS already exists, BASIC prompts:

```
File Exists--replace (Y/N):
```

Enter Y to execute the rename.

Care should be taken not to rename an open file, otherwise erroneous errors can occur on subsequent access to the channel on which the file was open, particularly on a network.

RENUMBER

```
RENUMBER      RENUMBER 100,20
RENUMBER 100  RENUMBER ,20
```

This command is used to tidy up programs after editing. All line numbers, and internal references such as GOSUB 115, are adjusted. RENUMBER may be followed by two arguments: the new first line number and the line number interval. If either is omitted the default of 10 is used. Thus the examples renumber the program from line 10 at intervals of 10, from 100 at intervals of 20 respectively, from 100 at intervals of 10, and from 10 at intervals of 20. Line numbers in REM lines, after MERGEGO, or in expressions (e.g. in comparisons with the ERL function described in Chapter 8) are not affected.

Extended BASIC Version 6 provides an additional facility to the RENUMBER command. An optional third argument, separated from the first two arguments by a comma, specifies the line number in the existing program from which renumbering is to start. (When only the first two arguments are used, renumbering begins at the first line of the existing program). For example:

```
Command:    RENUMBER 200,10,50
```

Initial program:

```
10 FOR I = 1 TO 10
20 GOSUB 50
30 NEXT I
40 END
50 REM Subroutine Square
60 LET J = I
70 LET J = J*J
80 PRINT J
90 RETURN
```

Resulting program:

```
 10 FOR I = 1 TO 10
 20 GOSUB 200
 30 NEXT I
 40 END
200 REM Subroutine Square
210 LET J = I
220 LET J = J*J
230 PRINT J
240 RETURN
```

This facility allows the partial renumbering of the program. It enables gaps to be inserted between any two existing program lines to provide space for additional lines of program.

In the example, note that the internal references on line 20 have been suitably modified by the RENUMBER command.

Before altering a program, the RENUMBER command validates the new line numbers to ensure that existing line numbers will not be overwritten. In such a case, the error message "Illegal function" is displayed and renumbering will not take place.

RESET
RESET string

```
RESET
RESET "ADC"
```

This command may be issued after changing a disc (on either a stand-alone or a network system) instead of having to issue an equivalent CP/M or CP/NOS command. Any data files currently open for input or output will be discarded.

A series of checks is built into the CP/M operating system to prevent inadvertent overwriting of programs and data if you accidentally change a disc while a data file is open for writing. If you do try to write to a changed disc you will get an error message of the form:

```
BDOS ERROR ON B: R/O
```

Do not press <RETURN> or you might lose your program. You can recover by pressing <CTRL/F>, which enters the Front Panel, followed by J>103<RETURN>. This restarts BASIC. Issue the RESET command, then inspect your program with the LIST command to make sure it is intact before proceeding.

On a network system the RESET command will be denied if another user has a file open on the disc drive(s) to be reset, and the optional string parameter can be used to specify the names of the drives to be reset. For example:

```
RESET "ADBC"
```

will attempt to reset logical drives A, D, B and C. In this case, a warning message is given for any drives for which the reset operation is denied.

If issued without a parameter, RESET will attempt to reset all drives, and will not produce any warning messages.

Note that RESET, RUN and LOADGO will cause any open files belonging to the user issuing that command to be discarded without being closed. This helps to avoid certain unnecessary errors, but does result in any output files that have not been closed being lost. (This does not apply to spooler files).

RUN

```
RUN
RUN 100
```

All variables are initialized and the program is started at the lowest numbered line if none is given, or at the specified line number (100 in the second example).

SAVE

```
SAVE "STATS3"     SAVE "B:STATS3.BAS"
```

The SAVE command saves a program on disc. Like the LOAD command, it must be followed by a program name, enclosed in quotation marks or held in a string variable or expression. When the program is saved, BASIC returns the prompt "Ready:".

On a stand-alone system it is important to check, before issuing a SAVE command, that there is a disc in the selected unit and to make sure that you have not changed the disc in the unit since starting BASIC. If you have, issue the RESET command before the SAVE command, otherwise you will get a BDOS Read Only error (see RESET).

CHAPTER 5
EDITING

The editing facilities described in this chapter are equally applicable to both stand-alone and network operation. This chapter deals in turn with the editing facilities available to the 380Z and 480Z computers.

Simple editing can be accomplished by retyping the corrected line, but use of the editing facilities of BASIC will usually be easier and more reliable. On the 480Z the function keys are used for editing and any text typed will automatically be inserted at the cursor position. With the 380Z editor <SPACE> and <DELT> are used to move the cursor along the line and insertion must be specifically requested.

380Z EDITING

This Section describes the use of the 380Z EDIT command.

EDIT `EDIT 55<RETURN>`

When the EDIT command is executed, the specified line (55 in the example) is copied into the edit buffer. Commands can be issued to modify the line, and to put the corrected line back into the program.

In Extended BASIC Version 6 the form EDIT. is allowed. This will edit the line last referred to (see Chapter 4).

While operation of EDIT is quite simple, it is rather different from most character- and line-oriented text editors, so learning EDIT can prove difficult. However, a familiarity with at least some of the commands, especially <SPACE>, <DELT>, <I>, and <RETURN>, does speed program development. Most of the commands do not reflect the typed characters, and some cause no change at all on the screen, which can cause confusion to novices.

EDIT maintains a pointer within the buffer. Under normal circumstances all of the characters to the left of the pointer are displayed on the screen, whilst those to the right are not. However, this simple scheme can get out of step following the <D> and <DELT> commands, so the current position of the pointer should always be kept in mind.

Many of the commands take an optional numeric argument, which causes the command to be repeated the specified number of times. The number, which should lie in the range 1-255, must be typed before the command.

There is no effect on the screen while a number is being typed. The numeric argument defaults to 1. In the detailed description of the commands below, such numeric arguments are denoted by "n". The commands are in alphabetical order: a short example appears at the end of this Chapter. Illegal commands are ignored.

RETURN <RETURN>

The line to the right of the pointer is displayed and the original line is replaced with the edited version. This is the normal way to end an edit session.

ESC <ESC>

<ESC> terminates any command sequence. In particular, it resets any partly-entered number to 1, and it terminates Insert mode.

nSPACE 6<SPACE>

Pressing the space bar will move the pointer right, reflecting the character it steps over. 6<SPACE> will move the pointer six characters, reflecting the skipped characters.

A <A>

Pressing <A> causes the current attempt at editing the line to be abandoned. The buffer is refilled from the original program line, and the pointer is reset to the beginning of the line. This command is useful if an edit has become confused.

nD <D> 7<D>

The n characters after the pointer are deleted. The deleted characters are echoed between backslashes (\). These characters may appear as a half (½) with some versions of COS, or as forward slash (/) with others.

E <E>

<E> exits the editor and implements the changes but the amended line is not typed out. This command was designed for use with very slow output devices in mind, and is probably not useful on the 380Z using the normal screen.

nFc `<F>3  3<F>A`

Find the nth occurrence of character c and move the pointer to just before it. If there are fewer than n occurrences of c, the pointer is moved to the end of the line.

H `<H>`

Everything to the right of the pointer is deleted and Insert mode entered. See <I> command for details of Insert mode.

I `<I>`

Enter Insert mode. Subsequent characters are entered directly into the buffer. Insert mode is terminated either by <ESC>, which merely terminates Insert mode, or <RETURN>, which leaves EDIT and updates the line. <DELT> will remove incorrect characters, even those to the left of the position of the pointer when Insert mode was entered. However, the line number cannot be changed. Insert mode can thus be used to delete or replace faulty text only.

nKc `<K>A  4<K>O`

Kill (delete) all characters from the pointer up to but not including the nth occurrence of character c. The pointer is left positioned just before this character. The <K> command has no discernible effect on the screen. If there are fewer than n occurrences of the character c to the right of the pointer then the remainder of the line will be deleted.

L `<L>`

The line to the right of the pointer is displayed and the pointer returned to the beginning of the line.

Q `<Q>`

The Edit is abandoned without implementing the changes made. Control is returned to command level (the "Ready:" prompt). This command is useful after an edit has become totally confused.

nR `<R>` `3<R>`

The n following characters in the buffer are replaced by the next n characters from the keyboard. <ESC> and <RETURN> are treated as in Insert mode, i.e. <ESC> aborts the R command and <RETURN> leaves EDIT and implements the changes. The other control characters and <DELT> are ignored.

X `<X>`

<X> moves the pointer to the end of the line and Insert mode is entered. In effect, new characters are appended to the line.

nDELT `<DELT>` `8<DELT>`

<DELT> will move the pointer to the left, backing the cursor over the skipped character. 6<DELT> will back over 6 characters. <DELT> is essentially the reverse of <SPACE>. Note that in Insert mode, <DELT> removes characters from the buffer. <DELT> will not move the cursor past the line number.

As an example of 380Z editing, we will change BROWN to RED in:

```
10 DATA THE QUICK BROWN FOX
```

In the following, only <SPACE>, <DELT>, <RETURN> and the <I> command are used:

User types	`EDIT 10<RETURN>`
Computer	`10`
User types	`20<SPACE>`
Computer	`10 DATA THE QUICK BROWN`
User types	`I<D><D><D><D><D>`
Computer	`10 DATA THE QUICK`
User types	`RED<RETURN>`
Computer	`10 DATA THE QUICK RED FOX`

The usual sequence in editing is:

1. Enter EDIT
2. Move pointer and enter Insert mode
3. Delete and/or insert
4. Exit from Insert mode and from EDIT

We recommend that you practice thoroughly with this subset of the EDIT commands before attempting to add the more complex ones to your repertoire.

480Z EDITING

This Section describes the use of the line editing capabilities of BASIC on a 480Z and the EDIT command.

Line Editing

It is possible to move the cursor backwards in the line to carry out alterations as long as <RETURN> has not been pressed. Non-printing characters are used to perform these editing functions. These control characters are obtained by holding down the CTRL key and typing a character but most of the commonly used characters are available on the function and arrow keypad. Some of the functions also require the use of the SHIFT key.

A plastic overlay is available to fit around the function and arrow keypad, showing the effects of the keys during line editing.

As characters are typed to BASIC, printing characters are entered into the buffer at the cursor position if there is room in the buffer. If there is insufficient room, BASIC sounds the beeper.

The following list describes the available editing functions:

<Left arrow>	Cursor left
<Right arrow>	Cursor right
<Up arrow>	Cursor up
<Down arrow>	Cursor down
<F1>	Delete character backwards
<F2>	Delete character forwards
<F3>	Find next character typed (backwards)
<F4>	Find next character typed (forwards)
<SHIFT/left arrow>	Cursor to beginning of line
<SHIFT/right arrow>	Cursor to end of line
<SHIFT/up arrow>	Delete line or restart edit in EDIT mode
<SHIFT/down arrow>	Delete line (from buffer only in EDIT mode)
<SHIFT/F1>	Delete to beginning of line
<SHIFT/F2>	Delete to end of line
<SHIFT/F3>	Delete to next character typed (backwards)
<SHIFT/F4>	Delete to next character typed (forwards)
<CTRL/M> or <RETURN>	Terminate line
<CTRL/Z>	Erase line, halt execution of program
<CTRL/L>	Clear screen, cursor bottom left
<CTRL/_>	Clear screen, cursor top left
<CTRL/E>	Toggle printer echo (see text)
<CTRL/P>	Toggle printer echo (see text)
<CTRL/F>	Enter Front Panel (after check)
<CTRL/C>	Enter operating system (after check)

This editing mechanism is available for both entry of program lines and for data entered in response to INPUT LINE or LINE INPUT. At all times except after the clear screen commands, the current state of the line is displayed, with the cursor symbol flashing over the character after the cursor position. If the cursor is at the end of the line it will flash or be steady depending on the state of the monitor. An attempt to move the cursor beyond the start or end of the line moves it to that boundary. The functions obtained by <F3>, <F4>, <SHIFT/F3> and <SHIFT/F4> require that a character is entered at the keyboard. For the find commands, the cursor is moved to lie on top of the next occurrence of the designated character in the specified direction. For the delete commands, characters are deleted up to and including the designated character.

During line editing in this manner, the printer echo process, set with <CTRL/E> or <CTRL/P>, is disabled and the effect of the WIDTH statement is suspended. If printer echo is set, the line is reflected to the printer after the terminating <RETURN> is pressed. The effect of WIDTH is only available for material typed by the computer, usually as the result of a program running. Pressing <CTRL/E> or <CTRL/P> during line editing switches the state of the printer echo flag which will be used after editing is complete, but no visible indication is provided for these keys.

It is possible for a program to alter the effects of the function and arrow keys with the PUT command. Although EDIT (q.v.) resets these keys to suitable values, normal line entry does not. Attempts to enter a line after a program has altered the function and arrow keys may therefore become confused. Strange effects will be produced if a window is defined which is too small to hold the line being edited.

The EDIT Command

The EDIT command can be used to alter a line after <RETURN> has been pressed. The format of the command is: EDIT <line number>

In Extended BASIC Version 6 the form EDIT. is allowed. This will edit the line last referred to (see chapter 4).

The specified line is transferred to the line editing buffer and the cursor is placed after the line number. Thereafter, editing proceeds much as for normal line editing. The only exceptions are:

<SHIFT/up arrow>	restarts edit with fresh copy of line
<SHIFT/down arrow>	abandons edit without affecting the line

The function and arrow keys are also initialized. <SHIFT/up arrow> will work until the line is terminated by either <CTRL/Z> or <RETURN>. The line number is treated like any other part of the line and can be altered in the same way.

Line edit mode is terminated by <CTRL/Z> or <RETURN>. On exit, the old line will be left intact and the new line inserted into the program, overwriting any existing line with that number.

CHAPTER 6

DATA TRANSFER COMMANDS

As described in Chapter 3, variables are the names of spaces in memory reserved for the storage of data, which may consist of numbers, strings of characters, arrays of numbers, or arrays of strings.

Enough string space should be reserved by a CLEAR statement to hold all the characters that will be stored in string variables during execution of the program.

Numeric and string arrays should be dimensioned with a DIM statement before use. If a DIM statement is omitted, or if a value is assigned to an array element before a DIM statement for that array is executed, BASIC assumes limits of 0 to 10 (11 elements).

For these reasons, it is usual for CLEAR and DIM statements to be among the first in a program. CLEAR should occur before DIM.

In Extended BASIC Version 6 the KILL command can be used to delete an array which is no longer required (see chapter 10).

CLEAR `10 CLEAR  20 CLEAR 2000`

All variables and arrays are cleared. If a number follows, as in the second example, space is reserved to store that number of characters in string variables. Thus the second example reserves storage for up to 2000 characters. If the number is omitted, the string area is not changed. When BASIC starts, space for 100 characters is reserved.

If a program runs out of string space during execution, the error message:

```
No string space
```

is displayed and program execution stops. You can discover the current size of string space by the statement:

```
PRINT FRE(X$)
```

and the current amount of free memory by the statement:

```
PRINT FRE(X)
```

Neither X$ nor X are altered by these statements. String space can be increased by the CLEAR statement until free memory becomes quite small, but some free memory will be needed for statements such as FOR and GOSUB, depending on the program.

Another form of the CLEAR statement is permitted in Extended BASIC, of the form:

```
CLEAR 200,,1024
```

in which the second number adjusts the size of cache memory. Cache memory is used in conjunction with machine language routines. Note the peculiar syntax with the two commas. These are required to allow for an extension in BASIC Version 6 (see Chapter 13).

DATA

```
50 DATA 5,4,3
20 DATA BLUE,RED      30 DATA "BLUE,RED"
```

Numeric or string constants can be specified in a DATA statement ready for assigning to variables by means of a READ statement.

Special care is needed if string data is to contain spaces or commas. The safest way to deal with this is to place the strings between quotation marks, for example:

```
20 DATA " GONE AWAY, BACK SOON"
```

is taken as a single string with a space before GONE.

Spaces may be inserted in a string without quotes after the first character, for example:

```
20 DATA GONE AWAY, BACK SOON
```

would be taken as two strings without a space before GONE or after the comma.

DIM

```
10 DIM A(20), B(30,5), A$(40), D(J)
```

Ordinary variables take the forms A, A7, X9 (numeric) or A$, A7$, X9$ (string). Array variables require a DIM statement to reserve space for them. For example, DIM A(20) would reserve 21 array elements A(0), A(1), A(2)...A(20). As shown in the example, a single DIM can dimension more than one array. A value would have to have been assigned to J before the DIM statement was executed. If an attempt is made to DIMension an existing array, the error message:

```
Re-dimensioned array
```

is produced and program execution halts.

In Extended BASIC Version 6 the KILL command can be used to delete an array which is no longer required (see chapter 10).

EXCHANGE

```
25 EXCHANGE A$,B$      35 EXCHANGE C,D(I,J)
```

The EXCHANGE command swaps the values of two variables.

The format of the EXCHANGE command is: EXCHANGE v1,v2

v1 — one of a pair of variables whose values are to be exchanged.

v2 — the other of the pair of variables whose values are to be exchanged.

Both of the variables must be of the same type.

An attempt to swap a constant value (e.g. EXCHANGE A$, "NO" or EXCHANGE X, 4) generates the message "Syntax Error".

A single array element (but not a whole array) may be used as either of the variables.

Example:

```
EXCHANGE F$(3), G$
```

The value of the fourth element (arrays have a base of zero) of the string F$ is exchanged with the value of the string variable G$.

Note that the swapping of values can be carried out by use of three assignment statements and another variable:

```
LET T$ = A$
LET A$ = B$
LET B$ = T$
```

The EXCHANGE command is, however, considerably faster, particularly where strings are concerned, as it is only the pointers to the strings that are swapped, instead of multiple copies of the strings being made.

LET

```
10 LET A=B+2      10 A$="HELLO"
```

The word LET is optional. The expression on the right hand side of the = sign is evaluated and the result is assigned to the variable on the left hand side. The reader is advised to think of assignment (LET) commands as:

```
A ← B+2
```

which should be read as "A becomes equal to the value of B plus 2", although, of course, the = notation must be used.

For example, the statement:

```
LET A=A+1
```

will increase by 1 the value stored in A.

If an attempt is made to assign a string to a numeric variable, or vice versa, the error message "Type mismatch" is displayed and program execution is halted.

For example, the statement:

```
LET A = "123"
```

is not acceptable.

READ

```
30 READ A,B,C      40 READ A$,B$
```

The READ instruction collects values from DATA statements and assigns them to the specified variables. Data is picked up from DATA statements in strict order. If there is more than one line of DATA, the lowest line number is taken first. Attempts to READ more data than exists will cause the error message:

```
Out of data at line n
```

where n is the line number of the READ instruction currently executing.

Numeric DATA can be READ into a string variable because it is also string data. String DATA cannot be READ into a numeric variable. Bearing this in mind, and noting that there is only one data pointer, the programmer must ensure that the DATA matches the READ statements exactly.

The data pointer can be placed at any position in the program by means of the RESTORE command (see below).

The READ # command, for reading information from a random access file, is described in Chapter 13.

RESTORE

```
80 RESTORE      50 RESTORE 100
```

The data pointer is reinitialized to point to the start of the data so that it can be used again. If no line number is given, RESTORE leaves the data pointer at the first DATA statement in the program. If a line number is supplied, as in the second example, it is left at the first DATA statement after the specified line.

CHAPTER 7

CONSOLE AND PRINTER INPUT/OUTPUT

This chapter describes INPUT and PRINT, the two main commands used in BASIC for reading the keyboard and outputting to the screen. Some variants of these are also described, together with a group of commands that help in formatting output. Consideration of data transfer to and from data files is deferred until Chapter 12.

INPUT

```
50 INPUT A      50 INPUT A,B
50 INPUT "ANSWER"; A$
```

The program pauses to allow the user to enter data at the keyboard which will be assigned to the specified numeric or string variables. When BASIC executes an INPUT command, a question mark is displayed on the screen. Taking as an example the program fragment:

```
10 INPUT A
20 PRINT A
```

the following would appear on running the program:

```
RUN
?
```

BASIC waits for the user to enter some data and press <RETURN>:

```
RUN
? 2<RETURN>
2

Ready:
```

If <RETURN> is pressed without any data being entered, either a numeric value of zero or a null (empty) string is assumed.

More than one variable can be specified in an INPUT command. In this case, the entered items should be separated by commas. If insufficient items are entered, BASIC prompts the user to enter the remainder with a double question mark. If too many are entered, the error message "*Extra lost" appears. Using the earlier example:

```
RUN
? 3,4
*Extra lost
3

Ready:
```

If a number is expected and characters which cannot form part of a number are typed in, the error message "*Invalid input" is displayed and all data for that INPUT statement must be retyped. Again using the example program:

```
RUN
? 1/4
Invalid input
? 0.25
 .25

Ready:
```

An explanatory string is allowed after INPUT, enclosed in quotation marks and terminated by either a semicolon (;) or a comma (,). It is printed before the program pauses, instead of a question mark if the terminating character is a comma, else as well as the question mark. If an INPUT statement is interrupted by entering <CTRL/Z> and the program then continued, the prompt is repeated. An alternative method of prompting is to use the PRINT statement (see below).

It is important that data should be entered from the keyboard in a form that matches the variables in the INPUT statement. Wrong data or too much data causes the error messages already described. Special care is needed when string data is required to contain commas or leading spaces. One way of dealing with this is to type the string between quotes. For example, the response:

```
? " GONE AWAY, BACK SOON"
```

is regarded as a single string with a space before GONE. The INPUT LINE instruction provides another way of achieving this.

Spaces may be inserted in a string without enclosing it in quotes, if they occur after the first character. Thus:

```
? GONE AWAY, BACK SOON
```

counts as two strings, each containing exactly one space. The spaces immediately after the question mark and the comma are ignored.

INPUT LINE `50 INPUT LINE "Please type your name: ", N$`

The INPUT LINE instruction differs from INPUT in that only a string variable can be specified. The whole of the entered data is transferred to the variable, including spaces and commas, without the need to enclose the data in quotation marks. If more than one variable is present, BASIC prompts for additional input with a double question mark.

INPUT LINE can be used in writing "safe" programs that are less likely to be stopped by invalid data; the process of validating the data can be carried out by programmed instructions rather than being left to the BASIC interpreter. This statement may take the alternative format LINE INPUT.

PRINT

```
50 PRINT      50 PRINT "ANSWERS ARE";A,B
```

Values of variables or messages are printed on the screen. The word PRINT is followed by a sequence of variables, expressions, or strings enclosed in quotation marks. Each item except the last in the sequence should have a semicolon or comma after it.

The instruction PRINT by itself merely outputs a blank line. A number or the result of a numeric expression is output with a space preceding it if it is positive, or a minus sign if negative, and with a space following it. A quoted string, string variable, or string expression is output with no spaces added.

When two numbers are separated by a semicolon, they are always separated by at least one space. For example, the program fragment:

```
10 LET A=2
20 PRINT A;A
30 PRINT -A;-A
```

would produce:

```
RUN
 2  2
-2 -2

Ready:
```

When two strings are separated by a semicolon, they are output without any intervening spaces:

```
10 LET M$="MARY"
20 LET J$="JANE"
30 PRINT M$;J$
RUN
MARYJANE

Ready:
```

Commas between items cause the items to be printed in columns spaced 14 positions apart. The form:

```
PRINT A,,B
```

is allowed and results in the value of B being printed at column 28.

A semicolon or comma after the last item suppresses the new line which would otherwise follow the printing of that item. Rather than being at the beginning of a line, the output from the next PRINT statement will start where the current one leaves off.

The symbol ? may be used as a synonym for PRINT. Its use in programs does not improve their readability but it is often convenient for inspecting the value of a variable in direct mode.

Output from the PRINT statement may be echoed to the printer by pressing <CTRL/P> or <CTRL/E>. See Chapter 2.

LPRINT

```
50 LPRINT
50 LPRINT "ANSWERS ARE";A,B
```

This instruction is the same as PRINT except that its output goes to the printer, not the screen. Note that the command CLOSE §2 is used to terminate output to the printer when BASIC is being used on a network. (See Chapter 12).

The next two instructions, SPC and TAB, are only allowed as part of a PRINT or LPRINT statement.

SPC

```
40 PRINT A;SPC(5);B
```

Extra spaces (5 in the example) are placed between the two printed values in addition to the space that follows A and the space, or minus sign, before B.

TAB

```
25 PRINT A;TAB(25);B
```

The TAB command causes the screen cursor or printhead to move to the indicated column. Columns are numbered starting from zero. In the example, the space or minus sign that precedes the value of B would be output in the 26th column, i.e. column 25.

If a TAB command indicates a position prior to the current cursor or printhead position, the current position is not changed.

POS

```
40 LET P=POS(0)
```

The POS function returns the current position of the screen cursor, in columns numbered from zero. In the example the cursor position is stored in variable P. The value of the function argument must be zero. The use of other values is described in Chapter 12.

LPOS

```
40 LET P=LPOS(D)
```

As POS, but returns the position of the printhead on the device accessed by the L commands. D is a dummy argument.

WIDTH

```
10 WIDTH 40
```

When BASIC starts, the screen width is considered to be infinite. Output that results in the cursor moving off the end of one line is continued on the next line with no special action being taken. The WIDTH command sets the logical width of the screen such that BASIC automatically begins a new line after the specified number of characters has been output. To return BASIC to its initial state, reissue the WIDTH command with a value of zero.

The requested width must lie in the range 14-255, or be zero.

LWIDTH

```
10 LWIDTH 72
```

As WIDTH, but sets the logical width of the printer. LWIDTH is initially set to zero, giving an infinite width. It may be reset to any value in the range 14 to 255, or zero.

NULL

```
10 NULL 3,0
```

After each line has been output to the console BASIC usually sends a carriage-return/line-feed pair of characters, in order to position the cursor at the beginning of the next line on the screen. The NULL command will cause BASIC to follow each CR/LF with a character, repeated the number of times specified by the first argument of NULL, the ASCII code of which is specified by the second argument of NULL. The example follows each CR/LF with three NULs (ASCII code 0).

On a 480Z or Varitext 380Z the command NULL 2,11 can be used to achieve downwards scrolling.

The NULL command is primarily intended to help users of certain printers, such as the Texas Instruments Silent 700 and some Teletype printers, which require time for the carriage to return before starting a new line. It has also been found useful for indentation or for printing in double-width mode.

LNULL

```
10 LNULL 3,0
```

As for NULL, but refers to the printer. Note that where a particular type of printer requires null characters, it is useful to send them with both LNULL and NULL to allow use of the <CTRL/P> echo facility.

USING

```
10 PRINT USING 20,B
50 CREATE #36,"NEWFILE",USING 60
```

Extended BASIC Version 6 provides means of controlling the format of certain output. See chapter 14 for details.

The USING instruction may be used only in a PRINT, LPRINT or CREATE command.

CHAPTER 8

PROGRAM CONTROL AND STRUCTURAL COMMANDS

Normally, program lines are executed sequentially, in numerical order. This chapter describes the commands available for altering this order.

UNCONDITIONAL TRANSFER

The following two commands transfer execution to the specified line number unconditionally.

GOTO `80 GOTO 150` `90 GO TO 180`

The GOTO command causes execution of the program to be continued at the specified line number. As shown in the examples, the forms GOTO and GO TO are both allowed. If a line with the specified line number does not exist, the error message:

```
Undefined statement
```

is displayed and the program is halted. A check is made that the line number terminates the GOTO statement.

GOSUB `100 GOSUB 250` `250 RETURN`
RETURN

Before executing a GOSUB command, BASIC notes its location. Program execution is then transferred to the specified line number which is usually the start of a subroutine. The last statement executed in the subroutine should be a RETURN command. When this RETURN statement is reached execution is transferred back to the statement following the GOSUB command.

The forms GOSUB and GO SUB are both allowed. If a RETURN command is encountered without a preceding GOSUB, the error message:

```
RETURN without GOSUB
```

is displayed and the program halts. A check is made that the line number terminates the GOSUB statement. If control is transferred with ON BREAK or ON ERROR or ON EOF, any active GOSUBs are left on the stack. This may cause problems if attempts are made to resume execution inside subroutines.

CONDITIONAL TRANSFER

The IF...THEN...ELSE construct allows the programmer to execute statements depending on particular conditions.

IF...THEN

```
10 IF A=2 THEN PRINT B
```

The keyword IF is followed by a Boolean expression. The keyword THEN is followed by any statement, which will be executed if the Boolean expression is true. Thus, in the example, the value of B is printed if the value of A is equal to 2, but not otherwise. In either case, execution continues with the next line.

A Boolean expression is any numeric expression, but is often a relational expression constructed from two numeric or string operands separated by a relational operator. The two operands must be of the same type, numeric or string. The available relational operators are:

=	Is equal to
<>	Is not equal to
>	Is greater than
<	Is less than
>=	Is greater than or equal to
<=	Is less than or equal to

The two-character operators may be written ><, =>, or =<, without change of meaning.

As mentioned in Chapter 3, Boolean expressions may be prefixed by the NOT operator and may be combined with the AND and OR operators. In Extended BASIC Version 6 the XOR, MOD, MIN and MAX operators can also be used. When a Boolean expression consists of a simple numeric expression, variable or constant, it is considered false if equal to zero, or true otherwise.

In the special case where the IF statement governs an unconditional transfer of control using GOTO, for example:

```
10 IF A=2 THEN GOTO 100
```

either "THEN" or "GOTO" may be omitted. The forms:

```
10 IF A=2 THEN 100
10 IF A=2 GOTO 100
```

are permissible, and have an equivalent effect to the previous example.

ELSE

```
10 IF A=2 THEN PRINT B ELSE PRINT C
```

If the Boolean expression following the IF keyword is false and an

ELSE clause is provided, it is executed. An ELSE clause extends from immediately after the ELSE keyword to the end of the line. If the Boolean expression is true and the THEN clause is executed, the ELSE clause is skipped. A THEN clause extends from immediately after the THEN keyword to immediately before the ELSE keyword, if present, or to the end of the line if not.

From the foregoing it will be evident that either the THEN clause, or the ELSE clause, or both, may include a compound statement, i.e. a series of statements separated by colons.

The following example should clarify the behaviour of IF...THEN...ELSE:

```
10 LET J=1 :LET K=2
20 IF 2>1 THEN LET J=2 :LET K=3
30 PRINT J,K
40 IF 2<1 THEN LET K=0 ELSE LET K=5
50 PRINT J,K

RUN
2              3
2              5

Ready:
```

SELECTION

Two commands are available which allow the next line executed to be selected from a list.

ON...GOTO `10 ON X GOTO 400,500`

ON is followed by a numeric variable or by an expression returning a numeric result. GOTO is followed by a list of one or more line numbers, separated by commas. The value following ON determines to which line number program execution will pass.

In the example, if X = 1, execution is transferred to line 400, and if X = 2, to line 500. If X less than one or greater than 2, execution continues with the next statement after line 10. If X is less than —65536 or greater than 65535, the error message:

```
Illegal function
```

is displayed and the program is halted.

ON...GOSUB `10 ON Y GOSUB 400,500`

The ON...GOSUB construct is similar to ON...GOTO, described above, except that the statements whose numbers are in the line number list are called as subroutines. Each should end with a RETURN. When the RETURN command is executed, execution reverts to the statement following the ON...GOSUB statement.

The same restrictions and defaults apply to the numeric value following ON as with ON...GOTO. The example calls the BASIC subroutine at line 400 if Y = 1, or at line 500 if Y = 2.

LOOP CONTROL

Extended BASIC provides two statements which aid loop construction. These are the FOR and NEXT statements.

FOR..TO..STEP
NEXT

```
FOR X=1 TO 10          FOR Z=10 TO 1 STEP -1
NEXT X                 NEXT Z
```

The lines between the FOR statement and the NEXT statement are executed repeatedly until the loop control variable has taken all the values defined by the FOR statement. The first such value is the value of the expression following the equals sign, and further values are obtained by adding the STEP value (or 1 if it is omitted) until the TO value is reached or exceeded. Thus, in the first example, X successively takes the values 1, 2, ..., 9, 10, and in the second Z takes the values 10, 9, ..., 2, 1. The variable in the NEXT statement must correspond to the variable in the FOR statement. If it does not, the error message:

```
NEXT without FOR
```

is produced.

The statements in the loop are always executed at least once, even if the initial value is greater than the final value. Thus, the program fragment:

```
100 FOR X=10 TO 1
110 PRINT X
120 NEXT X
```

produces

```
RUN
 10

Ready:
```

FOR loops may be nested to any depth. That is, constructs of the form:

```
100 FOR X=1 TO 10
110 FOR Y=1 TO 10
120 LET A(X,Y)=0
130 NEXT Y
140 NEXT X
```

may be used.

The variable in the NEXT statement may be omitted. In this case, the variable used in the most recently executed FOR statement is assumed. Several variables may be specified in a single NEXT statement, separated by commas. The effect is the same as specifying the variables in a series of NEXT statements. It is probably wisest to avoid both of these facilities, on the grounds of clarity and portability.

The size of the STEP on a FOR loop is checked and if it is zero the error message "Illegal function" is printed and program execution stops.

A problem exists because of the way Extended BASIC handles its nested FOR loops. Whenever a FOR statement is executed, BASIC checks to see whether the specified variable is already being used as a loop control variable. If it is, the stack is cleared of all FOR loops up to this point. This means the following program fails:

```
10 FOR X=1 TO 10
20 IF X=3 THEN 40
30 NEXT X
40 FOR Y=1 TO 10
50 FOR X=1 TO 10
60 PRINT X;
70 NEXT X
80 PRINT
90 NEXT Y

RUN
1  2  3  4  5  6  7  8  9  10

NEXT without FOR at line 90

Ready:
```

The FOR loop at line 40 is cleared when the FOR loop at line 50 is set up, because there is already an active FOR loop using variable X.

If control is transferred with ON BREAK or ON ERROR or ON EOF (see Chapter 12) any active FOR loops or GOSUBs are left on the stack. This may cause problems if attempts are made to resume execution inside loops.

ERROR HANDLING

The following commands allow user control of error situations.

ON ERROR `ON ERROR GOTO 100`

Normally, when an error occurs, BASIC prints an error message and execution halts. It is occasionally useful to control what happens in the event of an error, and ON ERROR does this. The command specifies the line which should be executed if an error occurs. An error cancels the ON ERROR switch, so a second error will cause BASIC to print the error message. Thus, it may be convenient to make the error handling routine contain an ON ERROR statement.

The form ON ERROR cancels the effect of a previous ON ERROR GOTO..., and normal BASIC error handling resumes. Note that after transferring control via ON ERROR, any current FOR loops or GOSUBs remain active.

Any ON ERROR flag is cleared when BASIC is re-entered via its restart address (normally 103 hex). This means that the program does not automatically resume execution.

RESUME

```
RESUME
RESUME 150 (V.6 only)
RESUME NEXT (V.6 only)
```

RESUME is used to resume program execution after an error has occurred and been trapped with ON ERROR. BASIC goes back to execute again the statement which caused the error. Some caution is clearly necessary with RESUME to avoid creating an infinite loop.

In Extended BASIC version 6, the RESUME statement may be followed by either a line number that indicates at which line the program is to resume, or by "NEXT" to continue execution from the statement immediately following the statement at which the error occurred.

If a RESUME statement is encountered and no ON ERROR trap has been set, the message:

```
RESUME without error
```

is produced and program execution stops.

ERR `LET B=ERR`

ERR returns the number of the most recent error. A list follows of error numbers and the identifying message associated with each one. Appendix B gives further information including likely causes.

In the following list certain errors are marked D, C, 6 or N.

Those marked D apply only to disc systems; cassette systems will generate Unknown Error for these numbers.

Those marked C apply only to cassette systems; disc systems will generate Unknown Error for these numbers.

Those marked 6 apply only to Extended BASIC Version 6; Version 5 will generate Unknown Error for these numbers.

Those marked N apply only to network systems; stand-alone systems will generate Unknown Error for these numbers.

Error Codes

1		External error
2		NEXT without FOR
3		Syntax error
4		RETURN without GOSUB
5		Out of data
6		Illegal function
7		Arithmetic overflow
8		Out of memory
9		Undefined statement
10		Subscript out of range
11		Redimensioned array
12		Can't divide by zero
13		Illegal direct
14		Type mismatch
15		No string space
16		String too long
17		String too complex
18		Can't continue
19		Undefined user call
20		Illegal EOF
21		Files different
22		Recovered
23		Name not found
24		Can't verify ASCII files
25		Can't merge internal files
26		Unknown error
27		Read error
28		RESUME without error
29		Record too long
30		Invalid unit number
31		Missing file name
32		No input file
33		No output file
34		Invalid device
35		Invalid file name
36		Write error
37		Wrong internal format
38		File not found
39	D	Directory full
40	D	No disc space
41	D	Output close error
42	C	ROM Pack Read Error
43	N	Read only file/disc
44	N	File in use
45	N	Too many files on Network
46	N	Not logged in
47	N	Unknown I/O error
48	6	No end to DEF FN/PROC
49	6	No start to FN/PROC END
50	6	Illegal FN/PROC exit
51	6	Format error
52	6	Too many files
53	6	Invalid record length
54	6	Invalid file type
55	6	Reading unwritten data
56	6	Record number too large
57	6	File open for reading only
58	6	Invalid func. for CP/M 1.4
59	6	File locked
OTHER		Unknown error

In future versions, further errors will be assigned numbers upwards from the current limit. For a full description of the meaning of these error messages, see Appendix B. In earlier versions of BASIC, error number 26 generated the message "Missing Statement Number". This error no longer occurs.

ERL `LET A=ERL`

ERL returns the number of the line on which the most recent error occurred. Note that BASIC will not RENUMBER any line number which is being compared with ERL, so beware of problems arising from using RENUMBER in programs containing such lines as:

```
IF ERL=200 THEN ...
```

ERROR `ERROR 3` `ERROR`

ERROR causes BASIC to react as if the specified error had just occurred. If no arguments are given, ERROR generates error 0, "Unknown error". Otherwise, the error is chosen from the list given for ERR.

KEYBOARD INTERRUPT CONTROL

ON BREAK `ON BREAK GOTO 1000` `ON BREAK`

In a manner similar to ON ERROR, ON BREAK traps an attempt to interrupt a running program with <CTRL/Z> from the keyboard, and redirects program execution to the specified line. The form ON BREAK disables this feature, as does responding to <CTRL/Z>. Some caution is advised with ON BREAK, as with a little effort it is possible to write a program which cannot be interrupted except with <CTRL/F> or <CTRL/C>.

The main use for ON BREAK is to tidy up any loose ends which may remain after interrupting the program. Such uses include issuing a TEXT command (see Chapter 11) to disable graphics, or call a machine-code subroutine (see Chapter 13), perhaps to disable interrupts or clear the High Resolution Graphics screen. The ON BREAK command is not really useful for disabling the interrupt check completely. As with ON ERROR, FOR loops and GOSUBs remain active after an ON BREAK, so caution should be exercised when attempting to restart a program.

CHAPTER 9

NUMERIC, STRING AND USER DEFINED FUNCTIONS AND PROCEDURES

Functions perform a series of numeric or string operations on the specified arguments, returning a result as their value. BASIC provides numeric functions, string functions, and functions that are defined by the user. This Chapter describes the majority of these.

The additional numeric functions, string functions, and the pre-defined variables available with Extended BASIC Version 6 are also described in this chapter together with the facility for programming user-defined multiline functions and procedures.

Some specialised functions involved in Input/Output and interfacing with Assembly Language programs are described in the appropriate sections.

NUMERIC FUNCTIONS

BASIC provides numeric functions to perform standard trigonometric and algebraic operations. These numeric functions are:

SIN	Return the sine of an angle
COS	Return the cosine of an angle
TAN	Return the tangent of an angle
ATN	Return the arctangent of a number
SQR	Return square root of a number
EXP	Return the constant e raised to a power
LOG	Return natural logarithm
INT	Return integral part of a number
ABS	Return absolute value of a number
SGN	Return sign of a number
RND	Return a pseudo-random number

SIN

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

SIN returns the sine of the angle A, expressed in radians.

COS

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

COS returns the cosine of the angle A, expressed in radians.

TAN

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

TAN returns the tangent of the angle A, expressed in radians.

ATN

```
10 LET P=4*ATN(1)
```

ATN returns the arctangent, the angle (in radians in the range —pi/2 to pi/2) whose tangent is the argument of the function. The example assigns the value of pi to P.

It is worth observing that degrees can be converted to radians by the relationship:

radians = (degrees/180) * pi

as in the statements:

```
10 LET P=4*ATN(1)
20 LET R=(D/180)*P
```

which assign the radian equivalent of D degrees to R.

In Extended BASIC version 6 the variable PI is predefined to take the value 3.14159 (see below).

SQR

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

SQR returns the square root of its argument, which may not be less than zero.

EXP

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

EXP returns the algebraic constant e (about 2.718282) raised to the power of its argument. If the argument is equal to 1.0, the value of e is assigned to B.

In Extended BASIC version 6 the variable EE is predefined to take the value 2.71828 (see below).

LOG

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

LOG returns the natural logarithm (log to the base e) of its argument, which must be greater than zero.

INT

```
10 LET A=INT(5.25)      10 LET B=INT(-3.4)
```

The INT function returns the greatest integer that is less than or equal to the value of its argument. The first example assigns 5 to A. Note that when the argument is negative, INT returns a number whose absolute value is greater than or equal to that of its argument. The second example assigns —4 to B.

A common use of the INT function is to round numbers to the nearest integer. Thus:

```
10 PRINT INT(34.67 + 0.5)
20 PRINT INT(-5.1 + 0.5)
```

prints the rounded values 35 and —5.

A general point to be made here is that small errors are often made in floating point calculations. This may mean that the result of a calculation may be (say) 0.99999... instead of the expected 1.00000..., and if this quantity is INTed, the result will be 0 instead of 1. This phenomenon may be the cause of some spurious results (see Chapter 3).

ABS

```
10 LET B=ABS(4)      20 LET C=ABS(-4)
```

The ABS function returns the absolute value of its argument. If the argument is positive, ABS returns its value. If the argument is negative, its value multiplied by —1 is returned. The examples assign 4 to both B and C.

SGN

```
10 LET A=SGN(-4)
```

The SGN function returns 1 if its argument is greater than zero, 0 if it is zero, and —1 if it is negative. The example assigns —1 to A.

RND

```
20 LET B=RND(1)
```

RND returns a pseudo-random number in the range 0 to 1. If the argument is greater than zero, the value returned is the next in the pseudo-random sequence. If zero, the value returned is the same as was returned by the previous call, and if the argument is negative, the random number generator is reseeded.

For example:

```
10 LET A=RND(-1)
20 LET B=RND(1)
30 LET C=RND(0)
```

assigns a number between 0 and 1 to A, a random number to B, and the same random number to C.

Differing negative values of the argument may initialize RND differently so if a predictable sequence is desired, the same value should be used each time. Differing positive values have no effect.

Note that random numbers can be forced into the range A to B by the formula (B—A) * RND(1) + A.

Some sample statements follow, illustrating the use of the RND function:

Example 1.

To obtain a random number in the range 10 to 15, we can use:

```
LET D=(15-10)*RND(1)+10
```

Example 2.

To obtain an integer in the range 10 to 14, the INT function can be used:

```
LET E=INT(5*RND(1)+10)
```

The same sequence of random numbers is generated each time the program is RUN. The RANDOMIZE command described in Chapter 10 resets RND to a truly random point in its sequence.

The additional numeric operators provided by Extended BASIC Version 6 are:

MAX
MIN
MOD
XOR

and descriptions follow of each of these.

MAX

```
10 LET D = 6 MAX 4
```

The MAX function returns the higher value of two numerical expressions.

In the above example, C is assigned the value 6.

The expression —10.76 MAX — 9.9987 returns —9.9987 (since —9.9987 has a higher value than —10.76).

In the following example:

```
10    A=15
20    B=A MAX 4*4
```

the value 16 is assigned to B.

MIN

```
10 LET F = 6 MIN 4
```

The MIN function returns the lower value of the two numerical expressions.

In the above example, F is assigned the lower value 4.

In the following example:

```
10 LET A = 3 MIN 38
```

the value 3 is assigned to A

The expression —98.76 MIN 0.8765 returns —98.76.

MOD

```
10 LET A = B MOD C
```

The modulus (modulo) is the number of distinct integers in a finite system of numbers. In the modulo 5 system the numbers are 0, 1, 2, 3 and 4. In this system large numbers are expressed by dividing them by the modulus until a remainder less than the modulus is obtained. The number 19 is 4 in the modulus 5 system. If a counter is modulo 5 and it is set to 4 an increment of 1 will result in it being reset to 0.

In Extended BASIC Version 6, a number can be expressed in terms of any modulus number system by means of the MOD function.

Examples:

6 MOD 5	returns 1
A = 6 MOD 5	assigns the value of 1 to A
7 MOD 7	returns 0
—4 MOD	3 returns 2
4 MOD —3	returns —2
.75 MOD .5	returns .25
1 MOD .5	returns 0

The expression: n1 MOD n2
is equivalent to: n1 — INT (n1/n2) * n2
except that n1 and n2 are only evaluated once (and provided that n2 is non-zero). If, for example, n1 is a call to a random number generator, the random number used in the evaluation of the expressions would be the same for both occurrences of n1.

XOR

```
10 LET A = B XOR C
```

The function n1 XOR n2 performs an exclusive OR operation on the integer values of the operands n1 and n2, after conversion to 16-bit binary values.

When an exclusive OR operation is applied to two binary digits (each digit being either 0 or 1), the result is 1 if either (but not both) of the digits is 1, otherwise the result is 0.

When applied to two 16-bit values, the exclusive OR operation is performed on each of the 16 corresponding pairs of bits.

For example, the expression 7 XOR 15 returns the decimal value 8. The 16-bit values for decimal 7 and decimal 15 are evaluated as:

```
0000000000000111      (7)
0000000000001111      (15)
```

and the exclusive OR operation produces the 16-bit value:

```
0000000000001000      (decimal 8)
```

Further examples:

```
2 XOR 2              returns  0
1.5 XOR 6.5          returns  7
7.5 XOR 9.5          returns 14

40    A = 21
50    B = 6
60    C = A XOR B
```

This example results in the decimal value 19 being assigned to the variable C.

PREDEFINED VARIABLES

Extended BASIC Version 6 provides predeclared variables PI and EE with predefined values. These variables can be incorporated in a program without the need for values to be assigned to them.

PI PI is a variable initialized to 3.14159
EE EE is a variable initialized to 2.71828

The values of PI and EE are given their predefined values at the start of program execution but can be changed subsequently during execution. Hence existing programs in which these variables are already defined will run using Extended BASIC Version 6 exactly as before.

It is possible that a program, written without due regard to the advice given in chapter 3, may contain as its first reference to variable EE, say, a statement such as:

```
30 LET EE = EE + 6
```

This inadvisable (but valid) statement would assign to EE a value of 6.0 under Extended BASIC Version 5, but a value of 8.71828 under Version 6.

STRING FUNCTIONS

BASIC provides string functions that allow certain string to numeric conversions and permit examination and modification of strings. String functions that return a string have a dollar sign ($) at the end of their names. String functions that return a number do not have a dollar sign.

General information about strings, string variables and string arrays is contained in Chapter 3. Strings may be concatenated (combined) by the addition operator. Thus:

```
10 LET A$="12"
20 LET B$="345"
30 LET C$=A$+B$
```

assigns "12345" to C$.

The string functions are:

ASC	Convert character to ASCII code
CHR$	Convert ASCII code to character
VAL	Convert string to number
STR$	Convert number to string (decimal)
LEN	Return length of string
RIGHT$	Copy a segment from a string
LEFT$	Copy a segment from a string
MID$	Copy a segment from a string
HEX$	Convert to hexadecimal

Additional string functions available with Extended BASIC Version 6 are:

INSTR	Searches one string for a specified substring
FIX$	Returns a string truncated or extended with blanks to a specified length
SPACE$	Returns a string containing a specified number of spaces
STRING$	Returns a specified string repeated a specified number of times.

ASC

```
20 LET B=ASC("A")
```

ASC returns the number which is the ASCII equivalent of the first character of the string argument. For example:

```
10 LET A$="ABC"
20 LET B=ASC(A$)
```

assigns 65 to B, 65 being the decimal ASCII code for letter A. The example in the heading does likewise.

CHR$

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

CHR$ returns the ASCII character whose numeric code is the argument. The value of the argument must lie in the range 0 to 255. The example prints the letter A on the screen. CHR$ is the inverse of ASC.

VAL

```
20 LET B=VAL("4.5")
```

VAL returns the number represented by the string argument. The string should contain a numeric constant in any of the standard forms described in Chapter 3. Note however that hexadecimal digits in the range A-F must be in upper case. For example:

```
10 LET A$="4.5"
20 LET B=VAL(A$)
```

assigns 4.5 to B, as does the example in the heading.

The conversion is terminated by a character which does not form part of a valid number. An error does not occur if no numeric characters are encountered. Thus VAL("1.2.3") returns 1.2 and VAL("ABC") returns zero. If the string is the null string, VAL returns zero.

STR$

```
20 LET B$=STR$(4.5)
```

STR$ returns a string representing the value of the argument, preceded by a single leading space if positive, or a minus sign if negative. The example assigns the 4 character string " 4.5" to B$. STR$ is the inverse of VAL.

LEN

```
20 LET B=LEN("ABC")
```

LEN returns the length of the string argument string in bytes. If string is the null string, LEN returns zero. The example assigns 3 to B.

LEFT$
RIGHT$

```
20 LET B$=LEFT$(A$,2)
30 LET C$=RIGHT$(A$,2)
```

These two functions return substrings consisting of a number of characters taken from the left and right-hand ends of their string arguments, respectively. For example:

```
10 LET A$="ABCDE"
20 LET B$=LEFT$(A$,2)
30 LET C$=RIGHT$(A$,2)
```

assigns "AB" to B$ and "DE" to C$.

The value of the second argument is the number of characters to extract. If it evaluates to zero or the string is the null string, then the null string is returned. If it is greater than the length of the string, the whole string is returned.

MID$

```
20 LET B$=MID$(A$,2,3)
```

MID$ returns a substring extracted from the string argument, starting at the character position specified by the second argument and consisting of the number of characters specified by the third argument. For example:

```
10 LET A$="ABCDE"
20 LET B$=MID$(A$,2,3)
```

assigns "BCD" to B$.

If the second argument is greater than the length of the string, the null string is returned. If the third argument is omitted, all the characters from the position specified by the second argument to the end are returned. If the second and third arguments together are greater than the length of the string, MID$ will return a string obtained by starting at the second argument and going to the end.

Extended BASIC Version 6 allows the MID$ function to appear on the left hand side of an expression as well as on the right hand side of an expression. When the MID$ operator appears on the right hand side of an expression, MID$ returns a substring extracted from the string argument. When the MID$ operator appears on the left hand side of an expression a substring of the string argument is replaced by a specified string.

```
20 LET MID$ (A$,3,5) = "ABCDEFG"
```

In the above example the first 5 characters of the string "ABCDEFG" will overlay 5 characters of A$ starting with the third character of A$. If A$ was originally "123456789" then A$ would become "12ABCDE89". The first argument of the MID$ operator on the left hand side of an expression is the string to be overlayed. The second argument specifies the character position from which overlaying is to take place, and the third argument specifies the number of characters (from the string specified on the right hand side of the expression) that are to be overlayed.

If the overlaying string contains fewer characters than specified by the third argument, the overlaying string is extended to this length with trailing blanks. If it contains more characters than specified by the third argument, the extra characters are ignored (as in the above example).

HEX$

```
LET G$=HEX$(12345)
```

HEX$ converts its numeric argument into a four character string containing the number expressed in hexadecimal. The example assigns "3039" to G$

INSTR

```
10 LET P = INSTR (A$,B$,C,D)
```

The INSTR function searches one string (A$ in the above example) for a specified substring (B$). The function delivers a numeric result (assigned to the variable P in the example) which indicates the position of the first character of the search string (B$) in the searched string (A$). If no match is found then the value 0 is returned. An optional parameter (C in the above example) specifies the character position from which the search is to begin (defaults to 1). A second optional character (D) specifies the number of characters to be searched (defaults to the whole string).

If the string to be searched for (B$) is a null string, 0 is returned. For example, INSTR ("1234","") returns 0.

Invalid search values for C and D also returns the value 0. For example, INSTR ("123", "2",20) returns the value 0.

Examples:

1.
```
LET Q$ = "123456789"
```

a)
```
LET R = INSTR (Q$, "456")
```
assigns the value 4 to R.

b)
```
LET R = INSTR (Q$,"654")
```
assigns the value 0 to R

2.
```
LET V$ = "1234512345"
LET K$ = "34"
```

a)
```
LET R = INSTR (V$,K$)
```
assigns the value 3 to R

b)
```
PRINT INSTR (V$,K$,6)
```
outputs the value 8

c)
```
LET R = INSTR (V$,K$,6,2)
```
assigns the value 0 to R

FIX$

```
10 LET E$ = FIX$ (C$,N)
```

The FIX$ function returns a string of a specified length formed from a specified string. In the above example, E$ is assigned a string N characters long formed from the original string C$.

If the length of the required string E$ is less then that of the original string C$ then C$ is truncated. String E$ is padded with trailing space characters when N is greater than the length of C$.

The error message "Illegal function" is given when N is greater than 255 or negative.

Examples (with the space character represented by "_"):

```
FIX$ ("123456789", 3) returns "123"

FIX$ ("123", 9) returns "123_ _ _ _ _ _"
```

SPACE$

```
10 LET A$ = SPACE$ (N)
```

The SPACE$ function returns a string containing N spaces. If N is greater than 255 or negative then the error message "Illegal function" is given.

Examples (with the space character represented by "_"):

```
SPACE$ (5) returns "_ _ _ _ _"

LET D$ = SPACE$ (2) assigns "_ _" to D$
```

STRING$

```
10 LET F$ = STRING$ (S$, N)
```

The STRING$ function returns a specified string repeated a specified number of times.

In the above example F$ is assigned the value of S$ repeated N times.

The error message "illegal function" is given if LEN(S$)*N is greater than 255, or N is negative.

Examples:

```
STRING$ ("ABC", 3) returns "ABCABCABC"

LET H$ = STRING$ (K$, 4) assigns to H$ the value of K$ repeated 4 times.
```

USER-DEFINED FUNCTIONS

The keyword DEF together with the special function prefix FN allows the user to define functions which can, for example, replace an identical sequence of operations that occur in several places in a program. Function names consist of the letters FN, followed by a valid variable name, e.g. FNA, FNC7.

In Extended Basic Version 5 a function must be defined before it is referenced, and the form of a function definition is:

DEF FNvnvn (arg) = expr

where vnvn is a valid numeric variable name which becomes part of the function name, arg is a valid numeric variable name which is used as a dummy argument, and expr is a numeric expression which is to be evaluated each time the function is called.

As its name suggests, a single-line function cannot extend beyond the end of one program line.

Once a function has been defined, it may be referenced ("called") by the use, in any expression, of:

FNnof (vne)

where FNnof is the name of the function to be called and vne is a valid numeric expression. Expression vne is evaluated and its value replaces the dummy argument in the function definition. Next, the expression in the function definition is evaluated, probably making use of the value of the dummy argument, and the resulting value replaces the function name in the calling statement.

For example:

```
10 DEF FNS(X)=X*X
20 LET A=3
30 LET B=FNS(A)
40 LET C=FNS(4)
```

assigns 9 to B and 16 to C. Function FNS returns the square of its argument.

Note that the dummy argument X in the above example is quite distinct from the variable X which might be used elsewhere in the program. A dummy argument should be considered as a place marker which indicates to BASIC merely where to insert the corresponding argument from the function call.

Function definitions can be placed anywhere in a program but in Extended BASIC version 5 the program flow of execution must pass through a function definition before it can be called. For this reason it is usual to put definitions at the beginning of a program.

In Extended BASIC Version 6:

- It is not necessary for the flow of execution to pass through the definition
- The number of arguments of a function is no longer limited to one
- String, as well as numeric, values are permitted for arguments
- The function may return either a numeric or a string value.

If a user-defined function returns a string value, the function name must be suffixed with the character "$". The following example of a string function:

```
10 DEF FNAZ$ (A$) = LEFT$ (A$,1) + RIGHT$ (A$,1)
20 LET D$ = "ABCDE1"
30 LET E$ = FNAZ$ (D$)
```

assigns "A1" to E$. Function FNAZ$ returns a two-character string consisting of the first and last characters of its string argument.

A function cannot be defined in immediate mode. A function may be redefined as many times as desired; the definition used is the one through which program flow most recently passed. If program flow has not passed through a definition (possible in Extended BASIC Version 6) and there is more than one function with the same name, then the definition used is the one at the highest program line number. This also applies to multiline functions and procedures (see below).

If the definition of a function contains an error that will cause, for example, a "Syntax error at line nnn" message at run time, the line number in the message refers to the line in which the function was called, not the line that is in error.

MULTILINE USER-DEFINED FUNCTIONS

Extended BASIC version 6 provides for the definition of multiline functions in addition to the single-line functions available with previous versions.

The form of a multiline function is:

```
DEF FNvvn (arg1, arg2, ....)
.
.
.
.   (function body)
.
.
.
FNEND expr
```

where FNvvn is the function name (vvn being a valid variable name), arg1, arg2, ... are the dummy arguments of the function, FNEND defines the end of the function, and expr is an expression giving the value of the function. For example:

```
100 DEF FNSUM(X,Y)
120 FNEND X+Y
```

is a simple multiline function to add two numbers together. The function name is FNSUM, it has two dummy arguments, X and Y, and returns the value of their sum. The names of the dummy arguments are enclosed in parentheses and separated by commas. As is the case with single-line functions, the dummy arguments in the function are quite distinct from any variables that may have the same name outside the function. The value returned by the function is that defined by the expression after the FNEND keyword.

A call to a multiline function is made in a similar way to single-line functions. For example, to call function FNSUM:

```
LET A = FNSUM(4,6)
```

would return the value 10, which is then assigned to the variable A. The actual arguments to the function (4 and 6 in the example above) are, as with the dummy arguments of the function, enclosed in parentheses, and there must always be the same number and type of arguments in a function call as the number and type of dummy arguments in the function definition.

It is possible to use the function in any expression. For example:

```
PRINT FNSUM(4,6)
```

would output 10 to the screen.

The end of a function is defined by the FNEND statement which includes an expression giving the result of the function. Multiline functions also offer the possibility of multiple exits. Suppose that it is required that the original function FNSUM never returns a value greater than 99 and that a total greater than 99 should return the value 99, the function can be defined as follows:

```
100 DEF FNSUM(X,Y)
110 LET Z = X + Y       'Obtain sum
120 IF Z>99 THEN FNRETURN 99
130 FNEND Z
```

The FNRETURN statement in line 120 indicates that the function is to exit immediately, passing control back to the expression from which it was called. As with FNEND, the FNRETURN statement can be followed by an expression that gives the return value of the function. A function definition can contain any number of FNRETURN statements, but there must only be one FNEND statement. Therefore calling the above function with values whose sum is greater than 99, e.g.

```
LET Q = FNSUM(76,59)
```

results in the value 99 being assigned to the variable Q, whereas:

```
LET Q = FNSUM(23,42)
```

results in 65 being assigned to the variable Q.

The new version of the function FNSUM (above) uses a variable Z that is not a parameter to the function. Using this variable inside the function means that the value that the variable Z had outside the function is lost. It is possible to declare Z to be a local variable to the function and hence preserve the value of the variable Z outside the function.

Local and Global Variables

Suppose the first line of a BASIC program is:

```
10 LET Z = 3
```

Line 10 has "declared" variable Z and "assigned" to it the value 3. Variable Z is said to be a "global" variable since any subsequent line of the program can use variable Z or modify its value.

```
10 LET Z = 3              declares global variable Z and assigns
                                to it the value 3
.
.
.
120 LET B = Z * Z         declares global variable B and assigns
.                               to it the value 9
.
.
.
320 LET Z = 27            assigns the value 27 to global
.                               variable Z
.
500 END
```

The first mention of a variable in a program (for example, variable Z in line 10 and B in line 120 above) declares the use of that variable. After declaration, a variable name may appear as part of an expression (for example, variable Z in line 120).

Variables Z and B are termed global variables since they may be referenced anywhere within a program after the line in which they are declared has been executed. Global variables may, for example, be referenced within single-line or multiline user-defined functions.

Multiline functions, however, offer the additional (optional) facility of declaring variables to be "local" to the function.

Declaring a variable to be local means that the variable is created separately from any variable of the same name declared outside the function. In this way the "scope" of a

local variable is similar to that of a dummy argument of a function and limited to use only within the function in which it is declared.

A variable is declared to be local by placing it in a list starting with a comma and the keyword LOCAL and following the dummy argument list of the function. For example:

```
DEF FNEXAMPLE(D),LOCAL LA,LB$
```

declares the variables LA (numeric) and LB$ (string) to be local to the function FNEXAMPLE (the variable D is a dummy argument of the function and hence also local). When the function is called, the values of global variables LA and LB$ (if they exist) are not accessible, while the local variable LA is given a new value of zero and LB$ a null string. On exit from the function, the values of the global variables LA and LB$ (if any) again become accessible, and the values of the local variables are lost.

To return to the example function FNSUM:

```
100 DEF FNSUM(X,Y)
110 LET Z = X + Y
120 IF Z > 99 THEN FNRETURN 99
130 FNEND Z
```

The use of variable Z inside the function means that its original value is lost. To ensure that the variable Z is purely local to the function, rewrite the function as follows:

```
100 DEF FNSUM(X,Y),LOCAL Z
110 LET Z = X + Y
120 IF Z > 99 THEN FNRETURN 99
130 FNEND Z
```

If this program also includes:

```
10 LET Z = 42
20 LET Q = FNSUM (53,21)
30 PRINT Z
```

then the value output at line 30 would be 42 (the original value of Z). Inside the function FNSUM, the local variable Z has, however, taken the value 74, as seen by inserting a PRINT Z statement at line 115.

There is no restriction on the number of variables that can be declared as being local, except that they must all be declared on the same line as the DEF statement. Both local variables and dummy arguments must be simple variables; they cannot be array elements. The actual arguments supplied in a call to the function may, however, be array elements. For example: A = FNSUM (X(1), X(2)) is valid.

The ability to declare variables to be local to a user-defined function allows a function containing only local variables to be used in any program regardless of the names of global variables within that program.

Functions Calling Functions

A user-defined function (multiline or single-line) may call other user-defined functions or procedures. For example, the following function adds three numbers together by calling the function FNSUM twice:

```
50 DEF FNADD(P,Q,R),LOCAL A,B
60 LET A = FNSUM(P,Q)
70 LET B = FNSUM(A,R)
80 FNEND B
```

The actual arguments can be any valid expression, including a function, so the above function could be rewritten as:

```
50 DEF FNADD(P,Q,R)
60 FNEND FNSUM(P,FNSUM(Q,R))
```

The main limiting factor on the number of functions calling each other is the amount of memory in the computer, although if string variables are being used there is a further limiting factor (see below).

It is also possible for functions to call themselves. This technique is known as recursion. A sample program will be found at the end of this chapter which includes the use of a recursive function in order to calculate the factorial of any given number.

Multiline Function Definition

To summarize, there follows a general definition of a multiline function. Elements of the definition enclosed in square brackets are optional.

```
DEF FNs1 [ ( [d1 [,d2...] ] ) ] [,LOCAL l1 [,l2...] ]
.
.
.  (function body)
.
[ FNRETURN [e1] ]
.
.
.  (function body)
.
FNEND [e2]
```

where:

FNs1	=	the name of the function
d1	=	1st dummy argument
d2	=	2nd to nth dummy argument
l1	=	1st local variable
l2	=	2nd to nth local variable
e1	=	the expression giving the value of the function
e2	=	as e1, but attached to the FNEND statement instead of the FNRETURN statement.

Further Notes on Multiline Functions

A multiline user-defined function definition must begin with a DEF statement and end with an FNEND statement.

A function definition cannot contain a procedure definition or another function definition.

The DEF statement must contain the name of the function followed, optionally, by any dummy arguments and local variables.

A function name comprises keyword FN followed by any valid variable name.

A function that is to return a string value must have a dollar sign ($) as the last character of its name.

Dummy arguments of a function must be separated by commas and enclosed in parentheses.

If a function has no dummy arguments, the parentheses in the definition are optional, but empty parentheses () must appear in function calls.

Local variable names must be separated by commas, and the list preceded by a comma and the keyword LOCAL.

On entry to a function, each local numeric and string variable has assigned to it a value of zero or a null string respectively. (So do not use the same name for a local variable and a dummy argument.)

Dummy arguments and local variables must be simple variables; they cannot be array elements. (Any actual argument supplied in a call to a function may, if required, be an array element).

Any variable referred to in a function that is named neither as a dummy argument nor as a local variable is a global variable.

A function definition must contain only one FNEND statement.

From the FNEND statement, control returns to the calling statement.

A function definition may contain one or more FNRETURN statements, which provide alternative returns of control to the calling statement.

The FNEND and FNRETURN statements must contain an expression giving the value of the function. The value of the expression must be of the same type as the function. (A procedure should be used if a returned value is not required).

A function is called by using its name in an expression, together with any arguments that the function requires.

A function call must contain the number and type of arguments corresponding to those in the function definition.

An argument to a function can be any valid expression that results in a value of the required type.

The value returned from a call to a function is the result of the expression in the FNEND or FNRETURN statement through which control was returned from the function to the calling statement.

A function may call procedures or other functions, including itself.

A single-line function can only have one dummy argument, and no local variables.

It is not necessary for the flow of execution to pass through a function definition before that function is called, but the command RUN must be executed before it can be called. This means that you cannot call a function in immediate mode without first running the program.

If two functions have the same name, then the last one to be defined is taken as the function of that name.

A function can be redefined, by the flow of execution passing through its definition, any number of times.

Spaces are allowed after the characters "FN" in a function name, FN END statement or FN RETURN statement.

USER-DEFINED PROCEDURES

Extended BASIC Version 6 provides for the definition of PROCEDURES, which are similar to multiline functions, but with the essential difference that procedures do not return any value. Their use is therefore to provide extra commands that can be used in a BASIC program, (as opposed to the use of functions in giving values to be used in expressions).

A simple form of a procedure is:

```
DEF PROCvvn (arg1, arg2, ...)
.
. (procedure body)
.
PROC END
```

where PROCvvn is the procedure name (vvn being a valid variable name) and arg1, arg2, ... are the dummy arguments of the procedure. For example:

```
100 DEF PROC DSQ (A, B)
110 PRINT (A-B)↑2
120 PROC END
```

will display on the screen the square of the difference between the two values supplied as arguments.

As is the case with functions, the names of dummy arguments used within a procedure are quite distinct from any similar names used for variables outside the procedure.

A call to a procedure is made simply by using its name, followed by the required number of arguments, of the required types, in parentheses. For example, calling the procedure PROCDSQ as follows:

```
40 PROC DSQ (12, 15)
```

will result in the value 9 being displayed.

The end of every procedure must be defined by a PROC END statement, but "early" exits can be made if required from a procedure by means of PROC RETURN statements. For example, procedure PROC DSQ might be extended to:

```
100 DEF PROC DSQ (A, B)
110 IF A<10 OR A>20 OR B<1 OR B>50 THEN GOTO 140
120 PRINT "Difference squared =";(A-B)↑2
130 PROC RETURN
140 PRINT "Unexpected values :";A;B
150 PROC END
```

The discussion earlier in this chapter under the heading "Local and Global Variables" applies to procedures as well as to multiline functions, and the DEF statement of a procedure can contain a list of local variables, introduced by a comma and the keyword LOCAL. Declaring variable names to be local implies that variables with the same names can exist outside the procedure with no danger of their values being inadvertently changed by the execution of the procedure, but also that no values can be transmitted into or out of the procedure by being stored in any of those variables.

A procedure need not have any arguments or local variables.

Another feature common to both procedures and functions is the ability of one function or procedure to call another, even to the extent of calling itself, known as recursion. For an example of a recursive procedure, see the "Tower of Hanoi" program near the end of this chapter.

Procedure Definition

To summarize, there follows a general definition of a procedure. Elements of the definition enclosed in square brackets are optional.

```
DEF PROCn1 [ ( [a1 [,a2,...] ] ) ] [,LOCAL v1 [,v2,...] ]
.
. (procedure body)
.
[PROCRETURN]
.
. (procedure body)
.
PROCEND
```

where:

PROCn1	=	the name of the procedure
a1	=	first dummy argument
a2,...	=	subsequent dummy arguments
v1	=	first local variable
v2,...	=	subsequent local variables.

Further Notes on Procedures

A user-defined procedure definition must begin with a DEF statement and end with a PROCEND statement.

A procedure definition cannot contain a function definition or another procedure definition.

The DEF statement must contain the name of the procedure followed, optionally, by any dummy arguments and local variables.

A procedure name consists of the keyword PROC followed by any valid variable name.

Dummy arguments of a procedure must be separated by commas and enclosed in parentheses.

If a procedure has no dummy arguments, the parentheses in the definition are optional, but empty parentheses () must appear in procedure calls.

Local variable names must be separated by commas, and the list preceded by a comma and the keyword LOCAL.

On entry to a procedure, each local numeric and string variable has assigned to it a value of zero or a null string respectively. (So do not use the same name for a local variable and a dummy argument.)

Dummy arguments and local variables must be simple variables; they cannot be array elements. (Any actual argument supplied in a call to a procedure may, if required, be an array element).

Any variable referred to in a procedure that is named neither as a dummy argument nor as a local variable is a global variable.

A procedure definition must contain only one PROCEND statement.

When the PROCEND statement is executed, control returns to the calling statement.

A procedure definition may contain one or more PROCRETURN statements, which provide alternative returns of control to the calling statement.

The PROCEND and PROCRETURN statements may not contain an expression. (A function should be used if a returned value is required).

A procedure is called by using its name as a command, together with any arguments that the procedure requires.

A procedure call must contain the number and type of arguments corresponding to those in the procedure definition.

An argument to a procedure can be any valid expression that results in a value of the required type.

A procedure may call functions or other procedures, including itself.

It is not necessary for the flow of execution to pass through a procedure definition before that procedure is called, but the command RUN must be executed before it can be called. This means that you cannot call a procedure in immediate mode without first running the program.

If two procedures have the same name, then the last one to be defined is taken as the procedure of that name.

A procedure can be redefined, by the flow of execution passing through its definition, any number of times.

Spaces are allowed after the characters "PROC" in a procedure name, PROC END statement or PROC RETURN statement.

USING STRINGS AS ARGUMENTS OR LOCAL VARIABLES

A number of problems may be encountered when using string variables either as arguments or as local variables, either if a large number of strings are being used or if functions or procedures using strings make a large number of calls to other functions or procedures that also use strings.

When a function or procedure is entered, the current values of any arguments and global variables with the same name as any local variables need to be preserved by the BASIC interpreter. Any of these values that are of string type are stored in special string variables, called temporary string variables. The number of temporary string variables is limited to sixteen. Beyond this limit, BASIC will give the error message:

```
String too complex
```

Therefore, if functions that have a large number of string variables as parameters are being used, it is possible for this error to occur when the function is entered. These temporary string variables are also used whenevaluating string expressions (e.g. those involving LEFT$, MID$, A$+B$ etc.) so that it is possible that expressions that work outside a function will not work within it if a large number of string parameters are being used.

ERROR MESSAGES

1. After a DEF FN... or DEF PROC... statement, finding either another DEF statement or the end of the program before the matching FNEND or PROCEND statement will result in the error message:

```
No end to DEF FNPROC
```

2. Attempting to execute FNEND, FNRETURN, PROCEND or PROCRETURN when not inside a function will result in the error message:

```
No start to FNPROC END
```

3. Presenting a different number of actual arguments to a function or procedure than there are dummy arguments in the definition will result in the error message:

```
Syntax error
```

4. Attempting to pass a string value into a numeric dummy argument, or vice-versa, will result in the error message:

```
Type mismatch
```

   This error is also given if an attempt is made to return a numeric value from a string function, or vice-versa.

5. Passing a large number of string values to a function or procedure, or nesting functions or procedures with string parameters to a large depth, will result (as described above) in the error message:

```
String too complex
```

Note that if an error, of any sort, occurs within a function or procedure, then the variables are left as they were at that point, i.e. any local variables will have their local values. Also any functions are left on the stack. It follows that it is not advisable to exit from a procedure or function except by the appropriate exit command.

This implies that if you wish to trap <CTRL/Z> or errors with ON BREAK or ON ERROR, you should transfer control to another area within the function or procedure, where you set a flag to indicate the error and return to thecalling statement before checking the flag.

The following example should clarify this:

```
100 DEF PROC STARS (N)
110 ON BREAK GOTO 180
120 ON ERROR GOTO 200
130 FOR I = 1 to N
140 PRINT "*";
150 NEXT I
160 BF = 0 : EF = 0          'Clear flags
170 PROC RETURN
180 BF = -1                  'Set BREAK flag
190 PROC RETURN
200 EF = -1                  'Set ERROR flag
210 PROC END
```

EXAMPLE OF A RECURSIVE PROCEDURE

One of the most famous examples of recursion is the Tower of Hanoi problem:

```
 10 REM - Program to solve Tower of Hanoi
    problem for three poles
 20 INPUT "Number of discs"; ND
 30 PROC HANOI (ND, 3, 1, 2)
 40 REM - Move ND discs from pole 3 to pole 1
    using pole 2
 50 END
100 DEF PROC HANOI (N, F, T, U)
110 REM - Move N discs from pole F to pole T
    using pole U
120 IF N=1 THEN PRINT USING 170; F, T :
    PROC RETURN
130 REM - If only one disc then move it
    and exit
140 PROC HANOI (N-1, F, U, T)
150 PRINT USING 170; F, T
160 PROC HANOI (N-1, U, T, F)
170 IMAGE Move disc from## to##
180 PROC END
```

The output for three discs should be as follows:

```
Move disc from 3 to 1
Move disc from 3 to 2
Move disc from 1 to 2
Move disc from 3 to 1
Move disc from 2 to 3
Move disc from 2 to 1
Move disc from 3 to 1
```

You can check the above by using coins or books. Try it with more discs.

SAMPLE PROGRAM USING MULTILINE FUNCTIONS AND A PROCEDURE

```
100 "PROCEDURE PROCPOW Display powers of a number
110 "
120 DEF PROC POW (N), LOCAL I, P
130 LET P=N
140 IF ABS(P)>6501 THEN PRINT "Too big":PROCRETURN
150 FOR I=2 TO 10
160 LET P=P*N
170 PRINT N;"to the power of";I;"=";P
180 NEXT I
190 PROC END
200 "
210 "
220 "FUNCTION FNYN$ - Obtain a Y or N response
230 "
240 DEF FNYN$(), LOCAL A$
250 LET A$=GET$() "Wait for a key depression
260 PRINT A$
270 IF A$="Y" OR A$="y" THEN FNRETURN "Y"
280 IF A$="N" OR A$="n" THEN GOTO 310
290 PRINT "Please answer Y or N : ";
300 GOTO 250
310 FNEND "N"
320 "
330 "
340 "Recursive FUNCTION FNFAC - Calc. factorial
350 "
360 DEF FNFAC(I) "I is no of the factorial to calc
370 IF I=0 THEN FNRETURN 1 "Factorial of 0 is 1
380 FNEND FNFAC (I-1)*I
390 "
400 "
410 "MAIN PROGRAM - Using the above funcs & proc
420 "
430 PRINT
440 PRINT "Demonstration of Procedure FNPOW ? ";
450 IF FNYN$()="N" THEN GOTO 490
460 INPUT "Enter a number : ",N
470 FNPOW(N)
480 "
490 PRINT: PRINT "Factorial calculation ? ";
500 IF FNYN$="N" THEN GOTO 540
510 INPUT "Number to have factorial calc : ",NF
520 PRINT "Factorial";NF;"is";FNFAC(NF)
530 "
540 PRINT: PRINT "Another demonstration ? ";
550 IF FNYN$="Y" THEN GOTO 410
560 END
```

CHAPTER 10

MISCELLANEOUS COMMANDS

This chapter discusses all the commands which find no logical place elsewhere in the manual.

STOP `STOP`

STOP causes the message:

```
Interrupted
```

to be issued, and program execution stops. Program execution may be resumed at the statement after the STOP command with CONT, provided that the program is not modified.

END `END`

Program execution is halted. Unlike many other BASICs, there can be any number of END commands, including none at all, and they can be placed anywhere in the program. However, good programming practice suggests that if an END statement is present, it should be the last statement. Program execution may be resumed at the statement after the END statement, if any, with the CONT command.

BYE `BYE`

BYE closes all files and exits BASIC, returning to the operating system. This is the best way to leave BASIC if the program involved file handling, although for most purposes <CTRL/C> is probably easier.

REM `100 REM THIS IS A REMARK`

REM introduces a remark or comment into a program. It has no effect on the program, except to make it bigger and slower, but a human reader may find a commented program easier to read. The whole of the line following the REM is devoted to the remark, including any colons. This means that a REM statement must be the last or only statement on a line.

' comment `110 T=1 'INITIALISE T:GOSUB20' SET UP INFO`

Extended BASIC Version 6 allows every statement on a program line (except an IMAGE statement) to be followed by a comment.

The comment is prefixed with an apostrophe (' = SHIFT/7) and is terminated by either a colon (:) or the end of the line.

A line containing only a comment is valid.

If a comment is to include a colon then the comment must be prefixed by REM and not by an apostrophe.

If a DATA statement is to contain an apostrophe comment then that comment must be prefixed by a colon.

The following example indicates the use of the apostrophe comment to label subroutine calls:

```
40 IF A=3 THEN GOSUB 60 'QUERY:GOSUB 70' REPLY
```

The apostrophe comment can also be used to create identation with line labelling. For example:

```
10 'LOOP1          :FOR I=1 TO 10
20 'LOOP2             :FOR J=1 TO 10
30 '                     :PRINT I*J
40 'EXIT LOOP2        :NEXT J
50 'EXIT LOOP1     :NEXT I
```

LVAR `LVAR` `LVAR #10`

LVAR lists the values of all the simple variables, on the console in the first example, to the output file in the second (see Chapter 12). The variables are output one per line. Array variables are not listed. The LVAR command is most useful for debugging purposes.

LVAR may be included within a program.

LLVAR `LLVAR`

LLVAR is the same as LVAR, except that the output goes to the printer device.

On a network, terminate the output with CLOSE #2 (See Chapter 12)

TRACE `TRACE 0` `TRACE #10,1`

The TRACE command enables or disables program tracing on the specified channel, (the console and the output file in the two examples.) While tracing is enabled, the line number of any executed statement is output enclosed in angle brackets.

For example, the program:

```
10 TRACE 1
20 REM TRACING NOW
30 TRACE 0
40 REM NOT TRACING
```

produces:

```
<20><30>
Ready:
```

In the TRACE command, the last or only argument is evaluated. If it is zero, then tracing is disabled. Otherwise it is enabled.

Allowable channels are 0 (console), 2 (printer) and any serial file output channel.

In Extended BASIC Version 6, the number of the line currently being executed can be displayed by pressing <CTRL/T>.

LTRACE `LTRACE 1`

LTRACE is the same as TRACE, except that the output goes to the printer device. The same effect is acheived by TRACE #2, 1.

On a network, terminate the output with CLOSE #2 (See Chapter 12).

RANDOMIZE `100 RANDOMIZE`

As mentioned in Chapter 9, the RND function returns a random number. However, the same sequence of random numbers is returned each time the program is RUN. The RANDOMIZE command sets the random number generator to a random point in its sequence.

KILL `100 KILL A$,C`

As mentioned in chapter 6 the KILL command can be used to delete arrays which are no longer needed. The example deletes the arrays A$ and C. Note that simple variables cannot be deleted. By a combination of KILL and DIM it is possible to set every element of an array to zero.

PUT 27,....

Escape sequences provide a means of controlling certain system parameters from within a program. They are mentioned here merely to draw attention to their existence. See Appendix F for details.

CHAPTER 11

LOW RESOLUTION GRAPHICS

This chapter introduces the low resolution graphics facilities on all 380Z and 480Z systems. (High resolution graphics facilities, for which special hardware options must be fitted, are described in chapters 15, 16 and 17.)

The computer screen consists of 24 lines of 40 or 80 characters each, depending on the hardware configuration and the mode of operation that has been selected. Normally, the whole of this is used for scrolling. However, Extended BASIC has a special graphics mode whereby the bottom four lines only are used for scrolling, and the top 20 lines can be used for graphics only.

Each character cell consists of six dots arranged in a rectangle three high by two wide, with each dot capable of being set on or off individually. The "on" dots of a particular character can be either grey or white. However, all "on" dots within one character cell must be of the same shade.

The top 20 lines of the screen can thus be treated as an array of dots 60 high by 80 or 160 wide, with each dot being either off, grey, or white. The bottom four lines can be used for scrolling or graphics, and can be treated as an array 12 high by 80 or 160 wide.

GRAPH `GRAPH` `GRAPH 0` `GRAPH 1`

The effect of the GRAPH command depends on the value of the expression following it. This expression can take the values 0, 1, 2, or 3, or it can be omitted. If the argument is negative or greater than 3 the error message:

```
Illegal function
```

is produced and program execution halts.

GRAPH 1, or GRAPH on its own, sets the scroller to use only the bottom four lines of the screen and clears the top 20 lines, in preparation for graphics. GRAPH 0, or TEXT (see below), restores the full screen scroller.

With the 380Z, GRAPH 2 and GRAPH 3 are provided to maintain compatibility with previous versions. GRAPH 2 "opens" the screen so that PEEK and POKE work. However, it also blanks the screen, resulting at best in an annoying flicker at the top of the screen. GRAPH 3 restores the screen to its normal state. GRAPH 2 and GRAPH 3 will not work when in 80-character mode. With the 480Z, GRAPH 2 results in a slight pause but beyond this neither GRAPH 2 or GRAPH 3 have any effect.

TEXT `TEXT`

TEXT restores the scroller to use all 24 lines on the screen. It is equivalent to the GRAPH 0 command described above, although it is somewhat faster and leads to clearer programs.

PLOT `PLOT 10,12,2` `PLOT 18,21,"HELLO, WORLD", 2`

The PLOT command is used for plotting points, characters, and strings on the screen, anywhere in the top 20 lines. The first two arguments to PLOT are the x and y coordinates of the item to be plotted, where (0,0) is at the bottom left of the screen and (79,59 or 159, 59) is at the top right. However, no error is caused when the plotted point lies off the screen, provided that both x and y are less than 65535 and greater than −65536. Values of x and y outside this range cause the error message:

```
Illegal function
```

and program execution stops. Values of the y coordinate in the range −12 to −1 signify points in the bottom four lines of the screen normally used for scrolling.

The third argument to PLOT, if present, is the item to be plotted. If it is a string, then the string is plotted character by character across the screen. Thus:

```
PLOT 28,27,"HI THERE"
```

would result in the message "HI THERE" appearing in the middle of the screen.

If the third argument to PLOT is a number or numeric expression, the item plotted is either a point or a character. Values 0, 1 and 2 signify point plotting; the specified point becomes off, grey, or white respectively. Values in the range 3 to 127 result in the specified ASCII character being plotted, while those in the range 128 to 255 display a "graphics character".

It is important to appreciate the difference between plotting a point (third argument value 0, 1 or 2), which will appear at exactly the specified screen coordinates, and plotting a character (third argument value 3 to 255), which is of size 2 points wide by 3 points high and will appear in the same place on the screen if the given coordinates specify any one of the six points that it occupies.

The following program shows the character set. Each character occupies one character cell, two points wide by three high, and for clarity in this display alternate cells (both horizontally and vertically) are left blank:

```
100 GRAPH: PRINT: PRINT: PRINT: TEXT
120 N=0: MX=127: Y=51
130 Y=Y-6
150 FOR X=14 TO 74 STEP 4
160 PLOT X,Y,N
170 N=N+1
180 NEXT X
190 IF N<=MX GOTO 130
260 PLOT 0,-9,"Press <H> for others,"
265 PLOT 42,-9," or <RETURN> to end"
270 R$=GET$()
280 IF R$<>"H" AND R$<>"h" THEN END
290 IF MX>127 GOTO 100
300 N=128: MX=255: Y=51
310 GOTO 130
```

The above simple version of the program will display the available characters, but the addition of the following lines to that program (pay particular attention to the line numbers) will considerably improve the appearance of the display as well as affording further examples of the use of the PLOT command:

```
110 PLOT 14,54,"0   2   4   6   8  10  12  14"
140 N$=LEFT$("    ",4-LEN(STR$(N)))+STR$(N)
145 PLOT 0,Y,N$
200 FOR X=0 TO 78 STEP 2
205 PLOT X,51,31: PLOT X,-3,31: NEXT X
210 FOR Y=-3 TO 51 STEP 3
220 PLOT 0,Y,25: PLOT 8,Y,25
225 PLOT 78,Y,25: NEXT Y
230 PLOT 0,51,18: PLOT 8,51,16: PLOT 78,51,17
240 PLOT 0,-3,15: PLOT 8,-3,19: PLOT 78,-3,20
250 PLOT 6,-6,"Low-resolution graphics "
255 PLOT 54,-6,"characters"
```

If the third argument to PLOT is omitted, BASIC uses the last specified value, or 0 initially.

On a 380Z the fourth argument to PLOT sets up attributes. Attributes are available only on an 80-character board, and on a 40-character board the attribute argument has no effect. On an 80-character board, the attribute argument is treated as an 8-bit number, of which the least significant four bits control the available attributes. These are:

Bit	Decimal value	Effect
0 :	1	Alternate ROM
1 :	2	Underline
2 :	4	Dim character
3 :	8	Reverse video

Thus, by adding the decimal values, a value for the attribute argument in the range 0 to 15 can be chosen to set any required combination of

attributes. For example, a value of 10 will select underlining and reverse video. A value outside the range 0 to 255 will result in the error message:

```
"Illegal function".
```

If the attributes are omitted, they default to zero (no attributes), unless the third argument is also omitted, in which case they default to the last attributes specified. The default attributes are initially zero.

The fourth argument is not used on a 480Z because the 480Z has no attribute bits available, and is provided only to maintain a degree of compatibility. Although the attributes argument has no effect it must still, if present, lie in the range 0 to 255.

As a simple example, the following program plots a sine wave on the screen, together with a heading. On a 380Z with the 80-character option its heading will be in dim reverse video:

```
10 GRAPH
20 FOR X=0 TO 79
30 PLOT X, 25*SIN(X/10)+26, 2
40 NEXT X
50 PLOT 30, 0, "SINE WAVE", 12
60 PLOT -1,0,0,0
70 TEXT
```

Line 60 in included merely to reset the attributes to their normal states. If this is not done then subsequent plotting, which will probably use the default attribute values of "those last set", may well give unexpected results.

LINE `LINE 10,20,2  LINE X,Y`

LINE draws a straight line from the coordinates of the last PLOT command or the endpoint of the last LINE command to the specified position. The x and y coordinates can take the same range of values as for PLOT. The line will be black, grey, or white if the third argument, which defaults to the last value given, is respectively 0, 1, or 2. Larger values will cause the line to be drawn in characters, as is the case with PLOT. Again as with PLOT, an attributes argument can be specified but it will only have an effect with a 380Z with an 80-character board. This effect will be the same as with PLOT. For example, the following program draws a box in the middle of the screen, in black, grey, and white in sequence:

```
10 GRAPH
20 FOR I=0 TO 2
30 PLOT 10,10,I
40 LINE 10,49
50 LINE 69,49
60 LINE 69,10
70 LINE 10,10
80 NEXT I
90 GOTO 20
```

POINT `LET A=POINT(30,30)`

POINT returns the intensity at the specified coordinates on the screen. If the character at the specified location is a graphics character (i.e. it is a pattern of dots), POINT will return 0, 1, or 2 depending whether the point is off, grey, or white. If the point contains an ASCII character, that character will be returned.

POINTS `A=POINTS(30,30)`

POINTS will return the value of the character stored at the specified point on the screen. If the point is not a graphics character, POINTS will return the same value as POINT. A graphics character will be returned as a number in the range 128 to 255.

ATTRIB `A=ATTRIB(X,Y)`

The ATTRIB function returns the attributes at the specified location. On an 80-character 380Z the number returned will lie in the range 0 to 15. In a 40-character 380Z, or in a 480Z, the result will always be zero. See the PLOT command for the meaning of the attribute bits.

As an example of the use of some of the above facilities (and some to be described in Chapter 12), the following program will save the contents of the screen on file, and then redisplay it. This scheme allows a library of pictures to be maintained on file and be displayed as required:

```
 10 CLEAR 1500
 20 CREATE #10,"PICTURE"
 30 QUOTE #10,0
 40 FOR I=0 TO 19
 50 LET A$=""
 60 FOR J=0 TO 39
 70 LET A$=A$+CHR$(POINTS(J*2,I*3))
 80 NEXT J
 90 PRINT #10,A$
100 NEXT I
110 CLOSE #10
120 GRAPH
130 OPEN #10,"PICTURE"
140 FOR I=0 TO 19
150 INPUT LINE #10,A$
160 PLOT 0,I*3,A$
170 NEXT I
180 CLOSE INPUT #10
190 TEXT
```

CHAPTER 12

FILE HANDLING AND EXTENDED I/O

As well as both taking input from the keyboard or reading it from DATA statements and sending output to the screen or printer, in Extended BASIC it is possible to transfer information to and from files stored on disc.

Extended BASIC Version 5 allows only one input and one output file to be selected concurrently. Multiple files may be processed in the same program by selecting them one by one, in turn.

In Extended BASIC Version 5, these data files are organized and processed sequentially; you must read the entire file to read the last item of data it contains.

Extended BASIC Version 6 allows the simultaneous use of up to 109 files, and, in addition to sequential files, provides facilities for the use of random access files; you can read any portion of a random access file without the need to start reading from the beginning of the file. Other additional file handling facilities available with BASIC 6 are described in Chapter 13.

This chapter describes the use of sequential files for both BASIC 5 and BASIC 6. If using Extended BASIC Version 6, then wherever channel number 10 is used you can also use channels 20 to 127.

You use sequential files in the same way that you use terminal input and output. A modified form of the INPUT command takes its input from file, while file output is via a modified PRINT statement. Sequential files allow you to manipulate larger amounts of data in a shorter time than is possible with terminal I/O to the keyboard and screen.

This Chapter describes all of the commands which are used in conjunction with sequential data files. These are mostly commands which we have already met, chiefly in Chapter 7, extended to redirect their input or output to the file channel. A channel specification consists of the hash or sharp character (#) followed by the channel number. For input commands, the channel number can be 0, to indicate the keyboard, or 10, to indicate the file. For output, possible channels are 0, the screen, 2, to indicate the printer, and 10, to indicate the file. The channel specification follows immediately after the keyword, for example:

```
PRINT #10,A,B,C
```

and is separated from any further items by a comma. Commands that accept channel specifications are described below. In all except OPEN, CREATE and CLOSE, the channel specification defaults to zero. Many of the commands can alternatively be preceded by L to indicate the printer device. Such commands must not be followed by a channel specification starting with #.

Some of the commands require a file specification, which may be supplied within quotation marks or in a string variable or expression. File specifications were described in Chapter 4. You may also use the file specifications "CON:", "LST:", "RDR:", or "PUN:" to connect the file channel to the keyboard or screen, printer, reader, or punch device, respectively. These are useful for testing, and RDR: is useful for transferring programs between computers.

Also described in this Chapter are the single-character commands PUT and GET, which allow I/O to either file or console on a byte-at-a-time basis.

GENERAL FILE-HANDLING COMMANDS

This section discusses the commands which perform general housekeeping functions, such as opening and closing files. They all use channel 10, the file channel. The RENAME and ERASE commands described in Chapter 4 also perform useful file-handling functions.

CLOSE `CLOSE #10` `CLOSE #2`

The CLOSE command closes the output file. The output buffer is emptied and the disc directory is updated. If this step is omitted, the contents of the file will be lost. The BYE command described in Chapter 10 and the CREATE command described below both automatically close the output file if necessary.

CLOSE without a channel number specified closes all output files. If a channel number is specified then only the channel specified is closed.

Any output to a printer on a network (e.g. LLIST, LPRINT, LLVAR, LTRACE, PRINT #2, PUT #2, DIR #2) will create a file on the network server which will be subsequently printed. The file will not be printed unless it is closed — CLOSE #2 performs this function.

CLOSE INPUT `CLOSE INPUT #10`

The CLOSE INPUT command closes the file from which reading has taken place (namely the input file — the file opened by the OPEN command).

CLOSE INPUT without a channel number specified closes all input files. If a channel number is specified then only the specified channel is closed.

On a network, input files *MUST* be closed after use. Failure to do so may have severe repercussions on other network users.

CREATE `CREATE #10,"RESULTS"` `CREATE #10,Q$`

CREATE creates the specified file and makes it ready for output. If the file already exists, the message:

```
File exists--replace (Y/N):
```

is produced. If Y or y is typed at the console, the old copy of the file is erased and the new one created. Otherwise, the program halts. If the disc directory is full, the message:

```
Directory full
```

is produced. You must erase a file or change the disc before retrying. Remember that the RESET command must be issued after changing a disc (see Chapter 4).

LOOKUP `LET A=LOOKUP("DATA")`

LOOKUP is a function which allows a program to determine whether a file exists. This is often useful when a program wishes to OPEN a file, as an attempt to OPEN a non-existent file causes an error. The following program fragment is an example:

```
10 INPUT "FILE NAME";A$
20 IF LOOKUP(A$)<>0 THEN 50
30 PRINT "THAT'S NO GOOD - TRY AGAIN"
40 GOTO 10
50 OPEN #10,A$
```

LOOKUP returns —1 if the file exists, and 0 if either it does not exist or the file specification is malformed.

OPEN `OPEN #10,"NAMES"` `OPEN #10,F$`

OPEN opens a data file for reading. The specified file is opened and the file system is initialized. If an attempt is made to OPEN a non-existent file, the error message:

```
File not found
```

is produced and the program execution stops.

N.B. Always use the CLOSE INPUT command (described above) after you have finished reading an input file.

INPUT AND OUTPUT COMMANDS

The commands so far introduced merely set up files; they cause no input or output. This section describes the ways in which characters can be read from or written to files.

INPUT `INPUT #10,A,B`

INPUT from a file is entirely analogous to INPUT from the console. When an INPUT # command is executed, a line of data is read from the file. This line of text is scanned for data items which are then assigned to the specified variables in exactly the same way as described in Chapter 7. If there are two or more data items per line, they *must* be separated by commas.

Lines of data in a file will be in the correct format for an INPUT # command if the program that wrote them made correct use of the QUOTE command (see below).

No question mark prompts are produced even if insufficient items are given — under these circumstances BASIC merely reads another line. The form:

```
INPUT #10, "PROMPT"; A$
```

is not allowed.

Attempting to read a file before executing an OPEN command results in the error message:

```
No input file
```

and attempting to read beyond the last line of a file results in the error message:

```
Illegal EOF
```

INPUT LINE `INPUT LINE #10,A$`

The INPUT LINE #10 command is exactly the same as the normal INPUT LINE command, except that the data is taken from file and not from the console. If no file is OPEN, the error message:

```
No input file
```

is produced. Reaching the end of the file without an ON EOF results in the error message:

```
Illegal EOF
```

GET

```
10 LET A=GET(123)
20 LET B=GET(#10)      30 LET C=GET()
```

GET is a single-character input routine, which returns the ASCII value of the next character from the selected input stream. Console input may optionally be timed, allowing interactive response without halting the program.

The function may be provided with zero, one, or two arguments. If there are no arguments, GET waits indefinitely for a single character to be input at the keyboard. If there is one argument, it can be either a channel specification, indicated by #, or a delay, measured in centiseconds (hundredths of a second). Thus, GET(500) waits for 5 seconds for a response from the console, and GET(#10) gets a character from the input file. If two arguments are present, they must be a channel specification after # and a delay, and the channel must specify the console. In each case, if no character is available by the end of the specified delay period, 0 is returned.

If the specified delay is —1, GET waits for whatever time was left over from the last GET. GET(—2) returns the remaining delay.

A few examples should clarify all this:

`GET()`	Read next character from keyboard
`GET(100)`	Wait 1 second for a character
`GET(#0)`	Read next character from keyboard
`GET(#0,100)`	Wait 1 second for a character
`GET(#10)`	Read next character from file
`GET(-1)`	Allow remaining time
`GET(-2)`	Return remaining time
`GET(0)`	Test keyboard, return character or zero

GET$

```
10 LET A$=GET$(#10)      20 LET B$=GET$(123)
```

GET$ returns the next sequential byte as a string of length zero or one. It is similar to CHR$(GET(...)), except that GET$ cannot return the amount of time remaining by GET$(—2). Also, if the specified time delay is exhausted, GET$ returns the null string (as opposed to a string with a NUL in it).

With both GET and GET$, no special note is taken of any control character, including <CTRL/C> and <CTRL/Z>. This means that <CTRL/Z> is not recognised as a console interrupt during a GET from the console, or as end-of-file from the file channel. Continued GETting from file will eventually result in EOF only when the physical end-of-file is reached. If <CTRL/F> is typed while GET is waiting for a character, the value 6 is returned and the Front Panel is not entered.

Further examples of programs using GET are given at the end of this chapter.

PRINT `PRINT #10,A,B`

This form of the PRINT command is the same as the normal form introduced in Chapter 7, except that the characters are sent to the file.

If no CREATE command has been executed for the specified channel, then the error message:

```
No output file
```

is generated. The messages:

```
No disc space
Directory full
Write error
```

may also be produced. "Write error" may be caused by the disc being full or by a hardware malfunction, or, on a network, it may also indicate no disc present in the drive. "No disc space" means the disc is full.

PUT `PUT 13,"HI THERE"` `PUT #10,A$`

PUT outputs the elements of the list of arguments to the selected channel. The list consists of a sequence of numeric or string expressions, separated by commas. A numeric expression represents the ASCII value of a single byte; a string is output character by character. The width, null and quote options which govern the behaviour of PRINT do not apply to PUT. Similarly it has no effect on the position of the print head that is returned by POS and used by TAB. Examples of PUT are:

```
PUT 12              Clear the screen
PUT #10,"HELLO"     Output HELLO to file
```

As with all commands which output to file, the following error messages are possible:

```
No output file
No disc space
Directory full
Write error
```

Further examples of PUT will be found at the end of this chapter.

PUT can be used to output "escape sequences" of characters, which can be used to control certain system parameters from within a progam, and are described in Appendix F.

DIR
LIST
LVAR
TRACE

```
DIR #10          DIR #2,"*.*"
LIST #10,25-99
LVAR #10         LVAR #2
TRACE #10,1      TRACE #2,0
```

These commands behave exactly as described in Chapters 4, for DIR and LIST, and 10, for LVAR and TRACE, except that their output is sent to file (#10) or printer (#2). An attempt to output to a file for which no CREATE command has been executed will result in the error message:

```
No output file
```

INPUT CONTROL COMMANDS

This section describes the commands available for controlling input files.

ON EOF

```
ON EOF GOTO 1000      ON EOF
```

The ON EOF command controls the behaviour of Extended BASIC when an attempt is made to read past the end of file with GET, INPUT, or INPUT LINE. Normally, the message:

```
Illegal EOF
```

is produced. However, after an ON EOF GOTO... command, control is transferred to the specified line number. For example, the following program prints out a file on the console:

```
10 OPEN #10,"DATA"
20 ON EOF GOTO 60
30 INPUT LINE #10,A$
40 PRINT A$
50 GOTO 30
60 CLOSE INPUT #10
70 END
```

The command ON EOF restores BASIC to its original state whereby end-of-file causes an error.

EOF

```
EOF
```

The EOF command causes BASIC to react as if the end-of-file had been reached. This will either cause transfer of control to a line number specified by an ON EOF command, or cause an error message.

PUT CONTROL COMMANDS

This section describes the commands which control the format of output.

NULL `NULL #10,3,0`

This command is the same as described in Chapter 7, except that a channel specification can be given. The example causes each CARRIAGE RETURN/LINE FEED sent to the file to be followed by three ASCII NULs.

POS `LET A=POS(10)` `LET B=POS(2)`

POS returns the position of the "printhead" of the channel specified by the argument. The examples assign the cursor position of the file to A and the printer head position to B. POS is more fully described in Chapter 7.

QUOTE `QUOTE #10,0` `QUOTE #0,34`

This command is used to change the format of output generated by PRINT commands on a particular channel.

The first argument is the channel number and the second argument is usually 0 or 34. If the second argument is 34, then all strings output by PRINT commands on the specified channel are enclosed in double quotes — the character which has an ASCII code of 34. Also, commas separating multiple items on a PRINT statement are output literally instead of causing columnar output. For example:

```
 90 QUOTE #0, 34
100 PRINT A$,B,C
```

might produce the output:

```
"HI", 4 ,-5
```

If the second argument is 0, then output from PRINT statements will revert back to the normal (default) format. The second argument may in fact be any ASCII character code, although ASCII code 34 (double quote character) is the most commonly used. This "quote character" is the one used to enclose all strings output by a PRINT command on the specified channel.

This command is particularly useful if output generated by PRINT commands is later to be read in by an INPUT command.

The QUOTE command must be executed after the CREATE command for that file and before any PRINT # commands that are to have their output formatted in this way.

WIDTH `WIDTH #10,132`

The WIDTH command is the same as described in Chapter 7, except that a channel specification can be given. The example sets the logical width of the output file to 132.

In Extended BASIC Version 6, further formatting can be done by use of the formatted output specification (see Chapter 14).

EXAMPLES 1-5

These programs are intended to be short examples to perform simple functions on sequential data files. They are far from optimal in many respects, data validation in particular having been largely omitted for clarity. We strongly recommend that you try out these examples at the computer.

Example 1

The first example creates a data file with an arbitrary number of records, each of which contains a string and a number. Input is terminated by typing the line END,0.

```
 10 CLEAR 200
 20 INPUT "OUTPUT FILE NAME";A$
 30 CREATE #10,A$
 40 PRINT "INPUT THE DATA, ENDING WITH 'END,0'
 50 QUOTE #10,34
 60 INPUT A$,A
 70 IF A$="END" THEN 100
 80 PRINT #10,A$,A
 90 GOTO 60
100 CLOSE #10
110 END

RUN
OUTPUT FILE NAME? STOCK
INPUT THE DATA, ENDING WITH 'END,0'
? PENCILS,17
? BALLPOINTS,23
? PAPER,9
? STAPLES,10000
? PAPER CLIPS,2000
? END,0

Ready:
```

Example 2

The next example prints out a data file on to the console, checking that the file exists first. The file must consist of a number of records each containing one string and one number. The file generated in Example 1 meets this requirement.

```
 10 CLEAR 200
 20 INPUT "FILE NAME";A$
 30 IF LOOKUP(A$)=0 THEN 20
 40 OPEN #10,A$
 50 ON EOF GOTO 90
 60 INPUT #10,A$,A
 70 PRINT A$,A
 80 GOTO 60
 90 CLOSE INPUT #10
100 END

RUN
FILE NAME? STOCK
PENCILS          17
BALLPOINTS       23
PAPER             9
STAPLES       10000
PAPER CLIPS    2000

Ready:
```

Example 3

The next example copies a file, of any format, to another. It performs no error checking at all. Note the use of INPUT LINE command to ensure that the input file is copied without modification.

```
 10 CLEAR 200
 20 INPUT "FILE NAME";A$
 30 INPUT "OUTPUT FILE NAME";B$
 40 OPEN #10,A$
 50 CREATE #10,B$
 60 ON EOF GOTO 100
 70 INPUT LINE #10,A$
 80 PRINT #10,A$
 90 GOTO 70
100 CLOSE #10
110 CLOSE INPUT #10
120 END

RUN
FILE NAME? STOCK
OUTPUT FILE NAME? NEWSTOCK

Ready:
```

Example 4

The next example allows the user to append information on to the end of an existing file, or to create a completely new file. It does this by checking first to see whether the file exists (line 30). If it does, it is renamed to TEMP and copied. Otherwise, the file is simply created. New records can then be added, each record consisting of a string and a number. The input phase is ended by typing END,0 in response to the prompt.

```
 10 CLEAR 200
 20 INPUT "FILE FOR APPENDING";A$
 30 IF LOOKUP(A$)=0 THEN 140
 40 RENAME "TEMP",A$
 50 OPEN #10,"TEMP"
 60 CREATE #10,A$
 70 QUOTE #10,34
 80 ON EOF GOTO 120
 90 INPUT #10,B$,A
100 PRINT #10,B$,A
110 GOTO 90
120 CLOSE INPUT #10
130 ERASE "TEMP"
135 GOTO 150
140 CREATE #10,A$
145 QUOTE #10,34
150 PRINT "INPUT THE DATA, ENDING WITH 'END,0'"
160 INPUT B$,A
170 IF B$="END" THEN 200
180 PRINT #10,B$,A
190 GOTO 160
200 CLOSE #10

RUN
FILE FOR APPENDING? STOCK
INPUT YOUR DATA, ENDING WITH 'END,0'
? ENVELOPES,100
? CRAYONS,47
? ERASERS,29
? END,0

Ready:
```

Example 5

The last, most complicated, file-handling example is a bare-bones stock control program. A master file with a .NEW extension is updated from a transactions file with a .TRN extension. Each record of both files contains the now familiar one string and one number. If the string in a record of the transaction file matches one of the strings in the master file, the number from the transaction file is added to the master number. Otherwise, a new record is added to the master file. At the end of the run, the master file is updated and the transaction file deleted.

Deleting the transaction file avoids the possibility that the same set of transactions will be processed twice. For safety, a back-up copy of the master file is always kept in a file with a .OLD extension, and the old .OLD file is not deleted at the end of the run until the new .NEW file has been verified as correct.

```
 10 CLEAR 1000
 15 QUOTE #10,34     :REM Enclose strings in quotes
 20 LET MX=100             :REM 100 records maximum
 30 DIM D$(MX),D(MX)
 40 INPUT "FILE NAME";F$
 50 LET OM$=F$+".OLD"
 60 LET NM$=F$+".NEW"
 70 LET TR$=F$+".TRN"
 80 LET TM$=F$+".TMP"
 90 OPEN #10,NM$               :REM Open new master
100 ON EOF GOTO 160
110 FOR K=1 TO MX               :REM Read in master
120 INPUT #10,D$(K),D(K)
130 NEXT K
140 PRINT "TOO MANY DATA ITEMS"
150 STOP
160 CLOSE INPUT #10
165 OPEN #10,TR$
170 ON EOF GOTO 310
180 INPUT #10,D$,D                :REM Get new info
190 FOR N=1 TO K-1
200 IF D$<>D$(N) THEN 250
210 LET D(N)=D(N)+D             :REM Update master
220 IF D(N)>=0 THEN 180
230 PRINT "STOCK ITEM ";D$(N);
235 PRINT " HAS GONE NEGATIVE"
240 GOTO 180
250 NEXT N
260 LET D$(K)=D$                :REM Add new record
270 LET D(K)=D
280 LET K=K+1
290 IF K>MX THEN 140
300 GOTO 180
310 CLOSE INPUT #10
315 CREATE #10,TM$            :REM Create FILE.TMP
317 QUOTE #10,34
320 FOR N=1 TO K-1          :REM Output new master
330 PRINT #10,D$(N),D(N)
340 NEXT N
350 CLOSE #10
360 OPEN #10,TM$               :REM Verify output
370 ON EOF GOTO 430
380 FOR N=1 TO K-1
390 INPUT #10,D$,D
400 IF D$<>D$(N) OR D<>D(N) THEN 430
410 NEXT N
420 GOTO 470
430 PRINT "NEW MASTER INCORRECTLY UPDATED"
```

```
440 PRINT "  PLEASE START AGAIN."
445 CLOSE INPUT #10
450 ERASE TM$                    :REM Erase temporary
460 STOP
470 CLOSE INPUT #10
475 ERASE OM$                  :REM ERASE OLD MASTER
480 ERASE TR$             :REM ERASE TRANSACTION FILE
490 RENAME OM$,NM$         :REM NEW MASTER BECOMES OLD
500 RENAME NM$,TM$       :REM TEMP BECOMES NEW MASTER
510 END
```

EXAMPLES 6-8

The last few examples cover the use of GET and PUT for simple tasks.

Example 6

This example covers the "bare bones" of a simple program to teach children how to spell. This program appears to contain a simple INPUT statement, but in fact it will only accept the word "HOUSE". It will ring the bell on the printer device, if one is fitted, when the word has been correctly input. Of course, a genuine program of this nature would be considerably more complicated:

```
 10 PUT "Type your word now : "
 20 LET H$="HOUSE"+CHR$(13)
 30 FOR H=1 TO LEN(H$)
 40 LET M$=MID$(H$,H,1)
 50 IF GET$()<>M$ THEN 50
 60 PUT M$
 70 NEXT H
 80 PRINT "THAT IS CORRECT !!!"
 90 FOR L=1 TO 10
100 PUT #2,7
110 NEXT L
```

Example 7

This example reads a file using GET$ and prints it on the console:

```
10 INPUT "FILE NAME";A$
20 OPEN #10,A$
30 LET C$=GET$(#10)
40 IF C$=CHR$(&1A) THEN 70 :REM TEST FOR E-O-F
50 PUT C$
60 GOTO 30
70 CLOSE INPUT #10
80 END
```

Example 8

The last example is a game to test the players ability in addition:

```
100 PUT 27,61,48,74,31  :REM 40-ch mode & clear
110 INPUT "No. of seconds for each answer : ",S
120 PRINT
130 PRINT "You will be given 10 addition sums,"
140 PRINT "with";S;"seconds in which to"
150 PRINT "answer each one."
160 PRINT: PRINT "End each of your answers by"
170 PRINT "pressing the RETURN key.
180 PRINT: PRINT "Are you ready? (Y/N) : ";
190 X$=GET$()
200 IF X$="N" OR X$="n" GOTO 610
210 IF X$<>"Y" AND X$<>"y" GOTO 190
220 GRAPH: RANDOMIZE : WR=0: CO=0
250 X=INT(RND(1)*100): Y=INT(RND(1)*100)
270 X$=RIGHT$("   "+STR$(X),3)
280 Y$="+"+RIGHT$("   "+STR$(Y),3)
290 PLOT 20,21,X$
300 PLOT 18,18,Y$
310 PLOT 20,15,"---"
320 PRINT: PRINT "Answer = ";
330 T=100*S :REM Set time allowed for answer
340 R=0
350 Z=GET(T)
360 IF T>0 THEN T=-1
370 IF Z=13 THEN 470
380 IF Z=0 THEN 430
390 IF Z<ASC("0") OR Z>ASC("9") THEN 350
400 PUT Z
410 R=R*10+Z-ASC("0")
420 GOTO 350
430 PRINT
440 WR=WR+1
450 PRINT "Sorry - out of time"
460 GOTO 540
470 PRINT
480 IF R=X+Y THEN 520
490 WR=WR+1
500 PRINT "Wrong"
510 GOTO 540
520 CO=CO+1
530 PRINT "Correct!"
540 PRINT "          ";CO;"out of";CO+WR
550 IF CO+WR<10 GOTO 250  :REM Stop after tenth
570 GRAPH: PRINT: PRINT "Another go? (Y/N) : ";
580 X$=GET$()
590 IF X$="Y" OR X$="y" THEN PRINT: GOTO 100
600 IF X$<>"N" AND X$<>"n" GOTO 580
610 TEXT
```

CHAPTER 13

MULTIPLE FILE CHANNELS AND RANDOM ACCESS FILES

The additional facilities described in this chapter are available only with Extended BASIC Version 6.

The use of sequential files with Extended BASIC Version 5 (& 6) is described in chapter 12. Up to 2 such files may be in use concurrently.

Extended BASIC Version 6 has additional facilities for using random access files (described later in this chapter) and for handling multiple files.

MULTIPLE FILE CHANNELS

Extended BASIC Version 6 allows as many as 109 files to be open simultaneously. The channel numbers used are 10 for serial files (as used with previous versions of BASIC) and from 20 to 127 for more serial and random access files.

Only one file can be open on any given channel except in the case of channel 10 which can have two serial files open (one for reading and one for writing).

The additional commands (and extensions to existing commands) available in Extended BASIC Version 6 for handling multiple files are described below.

CLEAR `20 CLEAR 200,4,1024`

The CLEAR command (see chapter 6) is extended to allow the specification of the number of files to be used.

The second argument, which is optional, (4 in the example) indicates the number of files to be used. A default value of 2 is assumed if the second argument is omitted, for example:

```
20 CLEAR 2000,,1024
```

As in previous versions of BASIC, the first (optional) argument reserves space for string variables (2000 characters in the example) and the third (optional) argument adjusts the size of cache memory used in conjunction with machine-language routines.

The extended CLEAR command reserves an appropriate amount of buffer space at the top of memory according to the number of files that will be open at one time. Thus the more files you use, the less space there is available for the program.

ON EOF # `30 ON EOF #36 GOTO 400`

The construct ON EOF (see chapter 12) is extended to handle multiple files. Optionally, a channel number may be specified (#36 in the example). If no channel number is specified in the command then #10 is assumed. If the optional branch statement (GOTO 400 in the example) is not included in the command, control reverts to the BASIC error-handling routines upon encountering an end-of-file condition. The effect of an ON EOF line is not activated until the line is executed. Any previously activated line number for a given file is superseded by the line number in the last ON EOF line for that file to be executed.

EOF # `80 EOF #38`

The EOF command generates an end-of-file condition on the specified input channel (see chapter 12). If no channel number is specified, a value of 0 is assumed.

CLOSE # `50 CLOSE #36`

The CLOSE command (see chapter 12) is extended to handle multiple files. Only one channel number may be specified with each CLOSE command (#36 in the example). If no channel number is given, all output files are closed. Note that execution of the BYE command (see chapter 10), closes all currently-open files, including #2 (the printer channel). Any random access file that has write access can be closed with CLOSE.

CLOSE INPUT # `70 CLOSE INPUT #29`

The CLOSE INPUT command (see chapter 12) is extended to handle multiple files in a similar way to CLOSE, described above. Any random access file can be closed with CLOSE INPUT.

RANDOM ACCESS FILES

Chapter 12 describes the use of sequential files for storing and retrieving information. A random access file is also used to store information for subsequent retrieval but is organized in a different and more versatile manner than a sequential file. The organization of a random access file makes it possible to read information from anywhere on the file — at random — without the need to start the retrieval process from the beginning of the file. A sequential file can only be accessed from the beginning and the information it contains can only be processed in sequence, either during its creation or during subsequent reading.

The random access facility is achieved by organizing a random access file into records. A record consists of a specific number of bytes. The number of bytes in a record is specified by the user and is known as the record length. There is no restriction on the range of record lengths that can be chosen for a random access file. Every record within any one random access file has the same length. Different record lengths may be chosen for different random access files. The choice of record length depends upon the type and amount of data to be stored in a record. Each numeric value requires 5 bytes and each string value of n characters requires n+2 bytes. Any combination of string and numeric values may be stored within a record.

The data retrieval and data storage commands for random access files (READ and WRITE commands respectively) include an argument specifying which record in the file is to be written to, or read from. A command to read record number 50 can be followed by a command to read record number 15 (for example) without the need to close the file, reopen it, and restart the reading process from the beginning (as would be necessary with a sequential file). Record numbering starts from 0 for the first record of the file.

Any number of random access files may be created and subsequently used, although not more than 107 may be used concurrently (see the section on Multiple File Channels at the beginning of this chapter).

Random access files are distinguished by their user-defined alphanumeric name. When a random access file is being used by a program, the computer system refers to the file by a numeric name, called the channel number (user-defined within the range 20-127). The association between the filename and the channel number is made within the file creation command RANDOM. The command RANDOM is also used subsequently to open an existing file (again any valid channel number may be specified), and random access files are closed by means of the CLOSE or CLOSE INPUT commands.

The five principal commands associated with random access files are:

RANDOM	to create a new (or open an existing) file
READ	to read data from a file
WRITE	to write data to a file
CLOSE	to close one or any files having write access
CLOSE INPUT	to close one or any files having read access

These commands, together with the random access file utility functions, are described in detail in the remainder of this chapter.

The RANDOM command, including its specified channel number and filename, will cause a file to be created if no file with that name currently exists. If a file with that name does exist, the RANDOM command will open the file.

When a new random access file is created on a disc, the first few bytes are reserved as a file header block. The information held in the file header block includes the record length of the file when created and the number of the last written record in the file, and will be needed by BASIC when the file is subsequently reopened.

To protect an existing file from accidental overwriting, the RANDOM command also includes an optional argument known as the opening mode. When a file is opened in read-only mode, the contents of the file cannot be altered. To modify the contents of a file, the file must be opened in read/write mode (the default setting of the optional mode argument).

On a network system several users may require simultaneous access to a file, and provided they all open the file in read-only mode there will be no conflict. If, however, a user wishes to update a file that other users might access, then the users competing for access must all "lock" the file while they access it. The is achieved by making use of another mode of the RANDOM command, known as "lock" mode, which is introduced under "RANDOM Command" and explained in greater detail under "File Locking & Unlocking".

RANDOM Command

Random access files are initially created and subsequently opened by the RANDOM command. The syntax for the RANDOM command is:

RANDOM #n, "filename" [,[READ/WRITE/LOCK] [,r]]

The first argument specifies the channel number (in the range 20 to 127).

The second argument is a character-string that specifies the name of the file. The name can be given as a string variable, a string constant, or a combination of both, and must be specified according to the rules for disc files given in the publications "380Z Disc System Users Guide", "480Z Disc System Users Guide" and "Network Station Users Guide". A file name can be fully qualified, with a logical disc drive name and filename extension (assumed "BAS" if unspecified). A RANDOM command will create a new random access file if no file with the specified filename currently exists on the disc. If a file with the specified filename does exist, the RANDOM command opens that file for processing.

If the file opened is not a random access file then the error "Invalid file type" will be given. Note that this can happen if a program error occurs before the file has ever been closed, as the file does not become identified as a random access file until it is closed. To avoid this problem, either erase the file before setting it up, create and close the file before using it, or close it when an error is encountered.

A third, optional, argument is used to specify the mode in which an existing file is to be opened. There are three possible modes in which a file can be opened. The WRITE mode permits information to be read from and written to the file, while the READ mode permits information only to be read from the file. Opening a file in READ mode on a network system prevents accidental writing to the file and also permits other network users to access the file. When a file is opened and no mode is specified, the default mode WRITE is assumed. A file cannot be opened

in WRITE mode by more then one network user, and, once opened in WRITE mode, cannot be opened by other network users in READ mode.

The third mode in which a file may be opened by the RANDOM command is LOCK mode, and this is of use only to users of a network system. A file opened in LOCK mode is automatically locked before and unlocked after any read or write operation, so that any number of users may have a file open for reading or writing (but only one at a time may access that file).

While a file is open in LOCK mode, any attempt by another user to open it in another mode will result in a "File in use" error.

Any attempt to access a file (opened in LOCK mode) with READ, WRITE, CLOSE, CLOSE INPUT or TYP while another user has the file locked will result in a "File locked" error. If you wish to take action on this error you should transfer control with an ON ERROR command to a section of your program which checks for a lock error and takes appropriate action. You can also use the LOCK function, described later in this chapter, to test whether a file is currently locked by another user.

The LOCK function and UNLOCK command (described later) allow a user to lock a file while executing a series of READ/WRITE commands.

Access to a file opened in LOCK mode will be slower than access to a file opened in either READ or WRITE mode.

A sample program including the use of a file in LOCK mode is given at the end of this chapter.

A file opened in READ mode must be closed by the CLOSE INPUT command. An attempt to close it with the CLOSE command will result in the error messasge "Invalid unit number".

The fourth argument (optional in the case of opening an existing file) specifies the record length required for the file. When the RANDOM command is used to create a file, a record length (in bytes) must be specified and this information is held within the file header block. If no record length (or a record length of less that three) is specified when a file is created, the error message "Invalid record length" is displayed. When an existing file is subsequently opened the record length given when the file was created is used.

The RANDOM command will operate correctly only if run under version 2.2 or later of the CP/M operating system. An attempt to execute RANDOM under an unsuitable version of CP/M will result in the error message "Invalid function for CP/M 1.4".

Examples of RANDOM

1. `RANDOM #41, "B:MYFILE.DAT", WRITE, 100`

   Open a file called MYFILE.DAT on logical disc drive B, using channel number 41, and with a record length of 100 bytes, in WRITE mode (allowing both data retrieval and data storage).

2. `RANDOM #74, "MYFILE", READ`

   Open a file called MYFILE.BAS on the current logical disc drive, using channel 74, and in READ mode (no data storage permitted).

3. `RANDOM #123,A$+".RAN",,200`

   Open a file using channel 123 with a filename formed from the contents of the string variable A$ with a filename extension of RAN and with a record length of 200 bytes. As no file opening mode is specified the file is opened for both reading and writing, since the default opening mode is WRITE.

4. `RANDOM #20, "DATAFILE", LOCK`

   Open a file called DATAFILE.BAS in LOCK mode.

Note: An attempt to open a serial file as though it were a random access file results in the error message "Invalid file type". An attempt to create a random access file without specifying a record length generates the error message "Invalid record length". Though it is not necessary to specify a record length when opening an existing random access file, should a record length be specified that does not match that on the file, the error message "Invalid record length" is generated.

A random access file cannot be read/written by the sequential file commands, e.g. INPUT #, PRINT #. If a sequential read/write is required then the READ # and WRITE # commands must be used. Also, an existing sequential file cannot be read using the READ # and WRITE # commands. Any attempt to use a read/write command on a file of incompatible type results in the error message "Invalid file type". An example of how to convert a sequential file to a random access file is given at the end of this section on Random Access files.

READ # Command

The READ command requests the transfer of a data record from a random access file into main memory for processing.

The syntax for the READ command is:

READ #channel-number [[,]@record-number] [,variable] [,variable]...

The argument #channel-number specifies the channel to be used. The file must be opened previously with this channel number.

The optional argument @record-number specifies the number of the record to be retrieved by the READ command. If omitted, the next record in the file is retrieved. The comma separating the channel-specification and the record-number may be omitted. A file can contain a record with record-number 0.

An optional list of variable names may terminate the READ command. This list of variables is known as the input list. The READ command assigns the values contained in the record it retrieves from the random access file to each of variables in turn until all the variables in the list have had a value assigned to them. Execution of the READ statement ends either when all the variables in the list have received values or when all the data in the record has been assigned to such variables.

The error message "Reading Unwritten Data" is displayed if the end of the record is reached before all the variables have been satisfied.

An attempt to read a string into a numeric variable or a number into a string variable will result in a "Type Mismatch" error message.

A READ command that specifies a record number but does not include an input list causes subsequent READ commands that include an input list but no record number to read data sequentially from the beginning of the initially specified record number. For example:

```
50 READ #21 @10
.
120 READ #21, A, B, C
.
180 READ #21, D, E, F
```

The READ command at line 50 causes the READ command at line 120 to assign the first, second and third values held in record number 10 to variables A, B and C. The subsequent READ command at line 180 will assign the fourth, fifth and sixth values of record 10 to variables D, E and F.

In this way a random access file can, if required, be read sequentially from a specified record number.

A network system user may wish to read a group of records from a file, and to prevent access to the file by any other user until all the records in the group have been read. The LOCK mode of the RANDOM command

is inadequate for this purpose since it applies the lock separately for each READ command, so, in addition to opening the file in LOCK mode, use must be made of a LOCK function and an UNLOCK command, described below.

A special character known as an end-of-record marker (EOR) is automatically stored on a random access file at the end of each record stored on the file. Similarly an end-of-file character (EOF) is stored at the end of the last record stored on a random access file. If reading sequentially from a random access file, encountering an EOR marker does not generate an error and input continues from the start of the next record. Encountering an EOF marker however, generates the error "Illegal end of file". The TYP function, described later in this chapter, provides a method for detecting the imminent input of an end-of-file marker.

A READ command that does not specify a record number and does not include an input list is not illegal but has no effect.

Examples of READ #

1. `READ #21 @32, X, Y, Z`

   Reads three numeric values from record number 32 of the file previously associated (by means of the RANDOM command) with channel 21, to variables X, Y and Z. An error will be generated if either there are not at least three data items at the start of record 32 or if any of the three data items are not numeric.

2. `READ #67 @42`

   Causes any subsequent channel 67 commands that do not specify a record number to input (or output) data sequentially from the beginning of record number 42. Sequential input continues until a READ (or WRITE) command that specifies a record number is executed.

3. `READ #72, A$, B$, C`

   Reads two string values and one numeric value from the file associated with channel 72. The position from which input takes place will be determined from the most recent channel 72 READ or WRITE command that specified a record number, together with the total number of variables on the input list of that and any intervening READ or WRITE commands for channel 72 (with no record number). If no prior READ or WRITE command for channel 72 has been executed, the input takes place from the first record on the file (record number 0).

WRITE # Command `10 WRITE #59, @7, P$, COUNT, "HELLO"`

Output to a random access file is by means of the WRITE command. The WRITE command specifies the channel number to be used for output by means of the argument #n where n indicates the channel number (59 in the above example).

A second, optional, argument indicates the record number to which output is to take place by means of the expression @r, where r is the record number (7 in the above example). Set r to 0 to write to the first record of the file. The comma separating the channel-number and record-number arguments may be omitted.

An optional list of expressions may terminate the WRITE command. This list of variables is known as the print list (P$, COUNT and "HELLO" in the above example).

The WRITE command stores the value of the each of the expressions in its print list on the random access file. If the total length of data to be written exceeds the record length as specified in the RANDOM command then the error "Record too Long" is generated. If the total length of data to be written is less then the record length then the value of the remainder of the record is undefined.

A WRITE command that specifies a record number but does not include a print list causes subsequent WRITE commands that include a print list but no record number to store data sequentially from the beginning of the specified record number. In this way a random access file can, if required, be written to sequentially starting from a specified record.

A network system user may wish to update a group of records in a file, and to prevent access to the file by any other user until all the records in the group have been updated. The LOCK mode of the RANDOM command is inadequate for this purpose since it applies the lock separately for each WRITE command, so, in addition to opening the file in LOCK mode, use must be made of a LOCK function and an UNLOCK command, described below.

A WRITE command that does not specify a record number and does not include a print list, although not illegal, has no effect.

Values returned by TYP (see below) and error messages described above under READ are not reliable if attempting to read an unwritten record. (They operate as described when an attempt is made to read an unwritten field within a record). This means that if you are writing random records in such a way that "holes" may remain in the file, then you must either be very certain that no read will ever be attempted on an unwritten record, or, much better, arrange to write something to every record. It is suggested that if you know the maximum size of the file, you should write a simple file-initialization program, otherwise, whenever

writing a new record to the file, test its record number against the value returned by FLEN (see below). If the new record number is more than one greater than the highest existing record number, fill the "gap" with dummy records.

Examples of WRITE #

1. `WRITE #21, @32, 10, 5*5, A`

   Writes three numeric values to record 32 in the file associated (by means of the RANDOM command) with channel 21. The values written are 10, 25 and the value of variable A. Each numeric value requires 5 bytes of storage within the record.

2. `WRITE #21, A$, "hello", C`

   Writes two string values and one numeric value to the file associated with channel 21. As no record number is specified, the data will be stored immediately following the data item last read from, or written to, the file. Record boundaries are ignored and writing continues until all the print list contents are stored.

   Each string value requires a storage space within a record two bytes greater than the length of the string.

3. `WRITE #21`

   This command has no effect.

CLOSE Command `10 CLOSE #21`

An individual random access file is closed by means of a CLOSE command that specifies its channel number.

If no channel number is specified, all files open for writing are closed. This includes any random access files opened in WRITE or LOCK mode.

Any files opened in READ mode must be closed by using the CLOSE INPUT command.

All files *must* be closed after use.

CLOSE INPUT Command `100 CLOSE INPUT #49`

Individual random access files opened in READ mode must be closed by means of the CLOSE INPUT command, which specifies the channel number of the file to be closed. Files opened in WRITE or LOCK mode can also be closed with the CLOSE INPUT command.

If no channel number is specified, all files opened with read access are closed. This includes any random access files.

FILE LOCKING & UNLOCKING

A network system user may wish to access a group of records in a file, and to prevent access to the file by any other user until all the records in the group have been dealt with. The LOCK mode of the RANDOM command is inadequate for this purpose since it applies the lock separately for each READ or WRITE command, so, in addition to opening the file in LOCK mode, use must be made of a LOCK function and an UNLOCK command, described below.

LOCK Function

The LOCK function takes the form:

LOCK (#n)

and, when invoked, it attempts to lock the specified file, returning a value of −1 (=TRUE) if successful, or 0 (=FALSE) if unsuccessful because another user has the file locked. For example:

```
100 IF LOCK (#25) THEN WRITE#25,@23,A$,C ELSE 100
```

would repeatedly try to lock the file until it was successful.

A file locked by means of the LOCK function stays locked, preventing any other user from accessing any records in that file, until the user that locked it executes an UNLOCK command.

UNLOCK Command

The UNLOCK command takes the form:

UNLOCK #n

and it should be noted that since this is a command and not a function the channel number specification is NOT enclosed in brackets, nor is any value returned.

An attempt to LOCK or UNLOCK a file that is not a random access file will result in an "Invalid file type" error. An attempt to LOCK or UNLOCK a file that was not opened in LOCK mode is ignored. The UNLOCK command has no effect on a file that is not locked.

RANDOM ACCESS FILE UTILITY FUNCTIONS

TYP

```
80 LET D=TYP(#43)
```

The TYP function returns a numeric value corresponding to the data type of the next data item to be read from the file associated with the specified channel number (#43 in the above example).

The numeric values returned for each of the various data types are:

1 Numeric Data
2 String Data
3 End-of-record marker (EOR)
4 End-of-file marker (EOF)
Any other value (usually 0) indicates Unrecognizable Data.

An end-of-record marker (EOR) is returned at the physical end of the record. Despite the fact that an EOR is returned by TYP, reading a data item in a sequential manner (i.e. READ with no record number specified) would read the first data item from the next record.

An end-of-file marker (EOF) is returned at the physical end of a file.

Unrecognized data is returned when an attempt is made to read an uninitialized area of the file. For example, attempting to read more data from a record than was written to it.

Code 0 will not necessarily be returned if an attempt is made to read an unwritten record — see suggestion under WRITE for avoiding this problem.

Example

This example illustrates the use of the TYP function to avoid errors that would result in program termination by checking the data type before performing the READ operation. A simple message is printed for each of the 5 data types detected; however, each of the 5 subroutines could be expanded to make further use of the data.

```
 10 IF TYP (#20) > 4 THEN GOSUB 100
 11 ON TYP (#20)+1 GOSUB 100,200,300,400,500
 .
100 REM Unrecognisable Data
101 PRINT "UNRECOGNIZABLE DATA"
102 RETURN
103 '
200 REM Numeric data
201 READ #20,N
202 PRINT "NUMERIC VALUE="; N
203 RETURN
204 '
300 REM String Data
301 READ #20, C$
302 PRINT "STRING VALUE="; C$
303 RETURN
304 '
400 REM End of record
401 PRINT "END OF RECORD REACHED"
402 RETURN
403 '
500 REM End of file
501 PRINT "END OF FILE REACHED"
502 RETURN
```

Note the use of the apostrophe form of comment on lines 103, 204, 304 and 403 to improve the legibility of the program.

FPOS

```
10 LET R = FPOS (#40)
```

The FPOS function returns the number of the record to which data is being written to, or from which data is being read, in the file associated with the specified channel number.

The following example returns the value 59 to variable A:

```
100 READ #20 @59
200 LET A=FPOS (#20)
```

FLEN

```
 40 LET S=FLEN (#72)
```

The FLEN function returns the size of the random access file associated with the specified channel number. The file size is expressed as the number of the record with the highest record number in the file. For example:

```
100 RANDOM #21, "TESTF", WRITE,20
110 PRINT FLEN (#21)
120 WRITE #21 @50,"HELLO"
130 PRINT FLEN (#21)
140 CLOSE #21
```

Line 100 creates a new file called TESTF with a record length of 20 bytes and associates the file with channel number 21.

The output from this example program is:

```
0
50
```

Line 110 prints the length of TESTF when created (0). After data has been written to record number 50, line 130 prints the number of the last record in the file (50). Since record numbering starts from zero, there are now 51 records in the file.

RLEN

```
10 LET R=RLEN (#84)
```

The RLEN function returns the record length of the file associated with the specified channel number. For example:

```
10 INPUT "FILENAME"; A$
20 RANDOM #100, A$
30 LET A=RLEN (#100)
40 PRINT "Rec length ="; A; "characters"
50 CLOSE INPUT #100
```

RPOS

```
25 LET NC = RPOS (#75)
```

The RPOS function returns the number of characters of free space left in the record. For example:

```
10 RANDOM #88, "FRED", WRITE, 80
20 PRINT RPOS (#88)
30 WRITE #88, @20, "HELLO FRED"
40 PRINT RPOS (#88)
50 CLOSE #88
```

When run, this program produces:

```
80
68
```

since at line 20 the record to be written is empty so has 80 characters of space available, and by line 40 a string of length 10 has been written (strings occupy 2+length characters) so the remaining space is now 68 characters.

Sample Programs

1. To create a Random Access file having the same data as an existing Sequential file.

```
 10 REM Convert a sequential file
 20 REM to a random access file
 30 OPEN #20,"SEQFILE"
 40 RANDOM #30,"RANFILE",WRITE,252
 50 ON EOF #20 GOTO 120
 60 LET RN = 0
 70 INPUT LINE #20,A$
 80 WRITE #30 @RN, A$
 90 LET RN = RN + 1
100 GOTO 70
110 '
120 CLOSE #30
130 CLOSE INPUT #20
140 END
```

The above program takes a sequential file (SEQFILE.BAS) and copies its contents into a random access file (RANFILE.BAS). The sequential file is assumed to hold one string value in each of its records. A record length of 252 characters is chosen for the random access file. This should be enough to hold the largest string read from the file (250 characters), with two characters allowed for the internal codes of the file, one character to indicate that it is a string, and one to indicate the length of the string. The transfer continues until the end of the sequential file is reached.

2. To re-try several times if attempting to update a file that might be subject to locking by other users.

```
 10 ON ERROR GOTO 200
 20 RANDOM #22, "RANDFILE", LOCK
 30 READ #22 @33, A$, C
 40 PRINT A$, C
 50 WRITE #22 @33, A$, C+1
  .
  .
200 REM - Error Routine
210 ON ERROR GOTO 200          'Reset error trap
220 IF ERR <> 59 THEN 300      'Jump if not LOCK error
230 EC = EC + 1                'Increment error count
240 IF EC <= 10 THEN RESUME    'Retry up to 10 times
250 PRINT "File lock error"    'Give error message
260 STOP                       'Stop execution
  .
  .
300                            'Handle other errors
  .
```

CHAPTER 14

FORMATTED OUTPUT

This chapter describes the facility available with Extended BASIC Version 6 which permits the user to define the precise arrangement of information for output to a printer, screen or sequential file by means of the formatting command PRINT USING.

The PRINT USING command includes (or specifies the number of a program line that includes) an image of what is to be output. This image contains one or more format items. A format item determines the place and shape of each item to be output.

Items to be output are specified in the PRINT USING command by means of an output list. Each output item corresponds to a format item in the image.

The PRINT USING command may also specify the channel number upon which output is to take place. If no channel number is specified then channel #0 (screen channel) is assumed.

The following examples illustrate the terminology associated with the PRINT USING command:

```
30 PRINT #2 USING "Value 1 =### Value 2=%%%",22,13.9
```

The statement on line 30 will output to the printer:

```
Value 1 = 22 Value 2=014
```

In this example the PRINT USING command specifies a channel number and is followed by a string expression ("Value 1 = ### Value 2=%%%") and an output list (22,13.9). The string expression contains two format items (### and %%%) each corresponding to an item in the output list (23 and 13.9 respectively). The format items, made up of format characters (# and % in the example), are used to define the position and layout of the output items. Note that both format items include space for only three characters to be output, a sign and two digits, and that the second output item (13.9) is rounded (to 14) to satisfy the format item. These features will be discussed in more detail below.

The above example could equally have been written as:

```
30 PRINT #2 USING 80,22,13.9
80 IMAGE Value 1 =### Value 2=%%%
```

and would produce the same output as before. The format string has been placed on a separate line (line number 80) preceded by the keyword IMAGE and the PRINT USING command now refers to this line (80). One advantage of specifying the format string as a separate IMAGE statement is that numerous PRINT USING commands can use the same format string.

Note that when using IMAGE, formatting begins immediately after the letter E. In the above example a space will be printed before Value 1 ... etc.

A third method of writing the above example is:

```
20 LET A$ = "###"
30 LET B$ = "%%%"
40 PRINT #2 USING "Value 1 ="+A$+" Value 2="+B$, 22, 13.9
```

A summary of this introduction to the PRINT USING command is given below and is followed by a detailed description of the various methods available for the formatted output of numeric and string values.

The syntax of the PRINT USING command is:

PRINT [#n] USING Line number/string expression [;output list]

where the channel specification (#n) and the output list are optional and the slash character indicates a choice of either a line number or a string expression.

A line number is any valid line number in the range 1-65529. The line referred to must contain an IMAGE statement.

A string expression is any valid string expression representing a format string.

A format string is a string made up of format items and literal characters. A format item is made up of format characters.

Format characters are # % . , * ↑ < > $ + - !
Literal characters are any other valid character except "

The output list, which, if present, must be preceded by either a semicolon or a comma, is a list of numeric or string expressions separated by commas.

The output list is terminated by either a colon or the end of the line. Carriage return and line feed characters are output after the last item of the output list unless the last character of the output list is a semicolon or a comma. (There is no distinction between a semicolon and a comma in this context, so columnar output is not available with PRINT USING).

If an output list is not present in the print statement then only literal characters up to the first format character are output.

If the format string contains no format items then the format string is output literally, followed by the error message "Illegal function".

The syntax of the line on which a format statement is held is:

```
IMAGE format string
```

with the format string starting at the first character after the E of IMAGE, and terminated by the last non-space character on the line.

The format string on the same line as the PRINT USING statement is terminated by the end of the expression.

If all output for a particular channel is required to be in the same format then a format string can be specified on the CREATE command. For example:

```
CREATE #22,"MYFILE" USING "###.##"
```

The above specifies that all output for channel 22 will have three places in front of the decimal point (including one for the sign) and two places after the decimal point. This can be extremely useful for generating columns of numbers with decimal point alignment without having to specify the same format string on each PRINT statement.

The above technique is not solely restricted to files. For example:

```
30 CREATE #2, USING 40
40 IMAGE Value 1 = ### Value 2 = %%%
```

formats all output to the printer (LPRINT is equivalent to PRINT #2). Any output channel is permitted, including 0 (the screen).

The format will stay in use until the file is closed.

The format string affects only output from PRINT and LPRINT; not from PUT.

It is possible to overide the effect of any CREATE USING by specifying a different format string on the PRINT statement.

FORMAT CHARACTERS

In the example at the beginning of this chapter the values 22 and 13.9 were output as two-digit integer values 22 and 14 by means of the format items ### and %%% (13.9 being rounded to 14 to comply with the format requirement of three character places as specified by %%%).

The characters # and % are two of the 12 special format characters used to define format items:

```
# % * $ + - . ↑ , < > !
```

These format characters are used to define both numeric format items and string format items but have different effects in each case. The definition of string format items is described later in this chapter.

For numeric format items the characters # % and * are used to represent a character place.

The characters $ + and — are floating characters representing currency and sign symbols and also represent one character place.

A dot (.) is used to show decimal alignment within a format item.

An up arrow (↑) is used to show exponentiation.

A comma (,) is used to show where a comma is to be inserted within a format item.

Format characters < and > (when present), always indicate the beginning of the next format item for both string and numeric output. In the case of string format output (described later in this chapter) they also indicate the required justification of the string.

NUMERIC FORMAT ITEMS

There are three types of numeric output (integer, floating point and exponent form) whose format can be defined by three types of format item (i-format, f-format and e-format). On output, all numeric values are right justified (with leading spaces, zeros or asterisks inserted as required).

i-Format Item

A format item containing neither a decimal point nor an exclamation mark is an i-format item, and will cause a numeric value to be rounded to the nearest integer and output without a decimal point, for example:

```
PRINT USING "Value 1 = ###";-21.6
output:       Value 1 = -22
```

Note that #, % and * characters must not be mixed.

f-Format Item

A format item containing a decimal point but not an exclamation mark is an f-format item, and will cause a numeric value to be output with a decimal point and rounded to a sepecified number of decimal places, for example:

```
PRINT USING "Value 2 = ###.##";-21.6
output:       Value 2 = -21.60
```

Character Replacement

Leading # characters in an i-format or f-format item will often not be needed for a sign or digit, in which case they will be output as blanks.

Unused % characters in a format item will be output as zeros, and unused * characters as asterisks.

In an f-format item, only # characters are allowed after the decimal point, where they will indicate the number of decimal places required and will always be output as zero if not needed for any other digit.

Combinations of #, % and * characters are not allowed in an i-format item or before the decimal point in an f-format item.

e-Format Item

A useful way of representing numbers, especially very large numbers, is by the exponent form of notation where a number is expressed as a coefficient times a power of ten, for example:

$$\begin{aligned} 312 &= 3.12 \times 10^2 \\ 16{,}794 &= 16.794 \times 10^3 \\ \text{or } 16{,}794 &= 1.6794 \times 10^4 \\ 0.02 &= 2 \times 10^{-2} \end{aligned}$$

Since some printers and screens do not provide for superscripts (for example 3 in 10^3) the E-form of notation is used

```
   312 =  3.12E+2
16,794 = 16.794E+3
16,794 =  1.6794E+4
  0.02 =  2.0E-2
```

The e-format item forces numbers to be expressed in E-form and permits a choice of the permitted range of coefficient and exponent.

An e-format item is denoted by a number of up-arrows at the end of the format string. The first of these reserves space for the output of the letter E, the second to reserve space for the mandatory sign (+ or —) and the subsequent up-arrows reserve space for the exponent. The mantissa is rounded to the nearest integer and output right justified with leading zeros.

An exponent of zero is always given a positive sign.

Value replacement in an exponent occurs as described under i- and f-format items. A minimum of three arrows is allowed. If less are present then the error Format error is generated (see section on Errors).

Examples of e-format:

```
PRINT USING "V1 = %%%%↑↑↑↑      V2 = %%↑↑↑"; 312, 312
output:      V1 = 0312E+00      V2 = 03E+2

PRINT USING "V3 = ###↑↑↑↑↑     V4 = #####↑↑↑"; 16794, 16794
output:      V3 =  17E+003      V4 = 1679E+0

PRINT USING "V5 = ##↑↑↑        V6 = %%.#↑↑↑↑"; 0.02, 0.02
output:      V5 = 2E-2          V6 = 02.0E-02

PRINT USING "V7 = ######↑↑↑↑   V8 = %%%%↑↑↑"; 10, 65000
output:      V7 =  10000E-03    V8 = 0650E+2
```

Comma Insertion

In general, commas are inserted in the output whenever one occurs in a format item. If no digits have been generated then no comma is generated. Instead character replacement occurs (see i-format items). For example:

```
PRINT USING "V1 = **,***  V2 = %,%,%";23,42
output:      V1 = ****23  V2 = 004,2

PRINT USING "V3 = $-##,#.###,#"; -1
output:      V3 =   $- 1.000,0
```

Sign Insertion

The plus (+) and minus (—) signs are used to indicate that a sign should be inserted if appropriate.

If the value is negative, then, regardless of any sign in the format item, a minus sign is generated. If the value is positive, and the format item contains a plus sign, then a plus sign is generated. If the value is positive and the format item contains a minus sign, then a space is generated.

If no sign is included in the format item, then the first %, *, or # is taken and used as if it were a minus sign, following the rules outlined above.

The sign characters are known as floating characters since they are always printed immediately before the first digit of the printed value.

Examples of sign insertion:

```
PRINT USING "V1 = ##   V2 = -%   V3 = +#",-3,1,2,
output       V1 = -3   V2 =  1   V3 = +2

PRINT USING "V4 = +#",-4
output:      V4 = -4
```

Dollar Sign

A dollar sign ($) precedes a sign if a dollar sign is included in the specification. The dollar sign floats in the same way as a sign character, so that it is printed immediately before the first character of the value.

More than one dollar sign in a format item results in the error "Format error". See "Errors", below.

Examples of dollar sign use:

```
PRINT USING "V1 = $##    V2 = $-##",1,-3,
output       V1 =  $1    V2 =  $-3
```

```
PRINT USING "V3 = $%%%  V4 = +$###",2,-5
output:       V3 = $002  V4 =   -$5
```

If the dollar sign is required to appear in a fixed position instead of floating, then it should be output literally by preceding it with an exclamation mark (see "Literal Output", below).

Large Numeric Output

An attempt to output a numeric value containing more digits than the space reserved for the value in the format item results in no value being output and the formatted output item being filled with exclamation marks, for example:

```
PRINT USING "VALUE = ###"; 123456
output:       VALUE = !!!
```

FORMATTED STRING OUTPUT

The format characters # % $ — + * , ↑ . each represent one character place. Additionally, the two special format characters < and > are associated with the output of string expressions, and are used to specify the required justification of the output string.

Strings are NORMALLY CENTRED within a format string, but an angle bracket can be used to specify a preferred justification:

- < Indicates that the string is to be output with left justification.
- > Indicates that the string is to be output with right justification.

Example:

```
PRINT USING "S1=#######  S2=<%%","HELLO,"HELLO"
output:       S1= HELLO   S2=HEL

PRINT USING "S3=>$-######","HELLO"
output:       S3=    HELLO
```

A second angle bracket in a format string starts a new format string with the appropriate justification. In this way two strings can be concatenated when output. An angle bracket takes up one character place. Character places not filled by the output string are replaced by spaces.

If an output string will not fit in the format item then the extra characters are lost from the right hand side of the string.

If, when the output string is being centred, there are an odd number of spaces to be justified then the extra space is added to the right of the string.

Further examples of formatted string output:

```
PRINT USING "S1 = ###   S2 = <#<%%%%","HI","HI","THERE"
output:       S1 = HI    S2 = HITHERE

PRINT USING "S3 = ###", "Hello"
output:       S3 = Hel
```

LITERAL OUTPUT

Any character which is not a format character is output literally as shown in the previous examples.

To output a character which is a format character (# % $ — + * , ↑ .) precede the required character in the format item with an exclamation mark (!).

The exclamation mark is used as a special character to indicate that the subsequent character is to be output literally.

Note that in order to output an exclamation mark literally it is necessary to precede it with an exclamation mark.

Examples:

```
PRINT USING "Percentage = +##!%  !-###!%"; 50, 20
output:       Percentage = +50%   - 20%

PRINT USING "####!!  !H!E###"; "WHAT", "LLO"
output:       WHAT!   HELLO
```

REPETITION

When the number of items to be output exceeds the number of format items in the format string, the format string will be reused, for example:

```
PRINT USING "VALUE ## = ####"; 1, 100, 2, 200, 3, 300
output        VALUE  1 =  100VALUE 2 = 200VALUE 3 = 300
```

If you want to include a space between repeated output items, you must include space(s) at the beginning or end of the format item:

```
PRINT USING "VALUE ## = ####  "; 1, 100, 2, 200, 3, 300
output        VALUE  1 =  100  VALUE 2 = 200  VALUE 3 = 300
```

TERMINATION

When all output items have been processed, output ceases following the output of the next literal string, if any. For example:

```
PRINT USING "## = V1  ## = V2  ## = V3", 1, 2
output        1 = V1   2 = V2
```

IMAGE

An image statement must be the first and only statement on a line. If execution reaches an IMAGE statement, other than by way of a PRINT USING command, then the whole line is ignored and execution continues from the following line.

ERRORS

If the line number specified in the PRINT USING statement does not contain an IMAGE statement, or does not exist, then the error "Undefined statement" is given.

If an error in the format string is found, such as the presence of mixed format characters of a different type in an i-format item, the output to the specified channel halts as soon as the error is encountered. At this point, the format string containing the error is displayed on the console (on a new line) and an up arrow (↑) is placed beneath the character at which the error was found, followed by the error message: "Format error".

For example:

```
20 PRINT USING "number = %%%%####";42.59
```

would result in the following being output:

```
number =
number = %%%%####
             ↑
Format error at line 20
```

If an ON ERROR trap has been set, then the format string is not output, and execution of the program is transferred to the line number specified on the ON ERROR line.

If the format error occurred during the execution of IMAGE statement, the line number given is that of the PRINT command and not that of the IMAGE statement:

```
20 PRINT USING 40,"HELLO"
40 IMAGE <$$$$$$$
```

The above generates the output:

```
<$$$$$$$
  ↑
Format error at line 20
```

If the format string contains no format items, an "Illegal function" error message is given when the end of the format string is reached.

CHAPTER 15

GETTING STARTED WITH HIGH RESOLUTION GRAPHICS

Each BASIC interpreter is supplied in three versions, distinguishable by their filenames. Extended BASIC 5 has a filename stem of "BASICS", and Extended BASIC 6 has a stem of "BASIC6". The stem is followed by nothing, "G" or "G2". The "G" and "G2" versions incorporate, respectively, Level 1 and Level 2 High Resolution Graphics support routines and to use these routines the computer must be fitted with the (optional) High Resolution Graphics Board. This chapter introduces Level 1 routines with examples, chapter 16 presents a summary in alphabetical order and chapter 17 describes Level 2 routines.

We suggest that you work through this introductory chapter at the computer. It is intended as a tutorial and you will probably only want to go through it once. After that, Chapter 16 and Appendix A can be used as required.

Before starting to work through the examples you should ensure that your computer has high resolution graphics hardware installed. You must also use a version of BASIC incorporating high resolution graphics support.

Some special terms are used. Here is a brief glossary:

Pixel — a picture element. The number of pixels in a picture depends on the resolution selected. The amount of memory used to store each pixel (in bits) governs the number of intensities or colours with which it can be displayed.

Resolution — HRG can operate in high resolution (320 by 192 pixels per picture), medium resolution (160 by 96) or, on the 480Z only, extra high resolution (640 by 192).

Intensity — the intensity range of each pixel depends on its size in bits. For example, with 2 bits/pixel, its intensity can be defined symbolically as 0, 1, 2 or 3.

Colour — a term used to describe the physical brightness that corresponds to a given intensity. It can range from 0 to 255 (black to white). In colour systems red, green and blue components can be specified.

Colour lookup table — a part of the HRG hardware in which each intensity is mapped to its corresponding colour.

Page — two separate pages of graphics memory (two separate pictures) are available in medium resolution.

View — by restricting the number of bits/pixel it is possible to have more than one picture on each page. These "logical" pages are called views.

References to Extra High Resolution Graphics (EHR) apply only to the 480Z.

GETTING STARTED

Enter the following BASIC program:

```
10 GRAPH : TEXT
20 CALL "RESOLUTION",0,2
30 CALL "PLOT",0,0,3
40 CALL "LINE",150,90,3
```

and type RUN. A diagonal line should appear on the screen running from the left near the bottom to the middle. If you get an error message, type LIST and examine the program for typing errors. Pay special attention to the quotation marks surrounding the name of each CALLed routine and to the comma following. Note that all HRG subroutine names must be given in UPPER CASE characters. (BASIC converts, if necessary, all keywords and variable names, but always leaves text strings unchanged):

```
call "RESOLUTION",0,2   is valid
CALL "RESOLUTION",0,2   is valid
call "resolution",0,2   is invalid
CALL "resolution",0,2   is invalid
```

If all is well, you are now ready to start programming with high resolution graphics. If you find the text going too fast, feel free to pause and try some experiments with what you have learned so far.

CLEARING THE SCREEN

Now have a look at the simple graphics program you have just typed in (type LIST). Notice that the program text and the high resolution display are superimposed. In line 10, the command GRAPH clears the screen of text before plotting and confines subsequent text output to the bottom four lines of the screen, below the graphics area. TEXT allows text to occupy the full screen again. If you are unfamiliar with the effects of GRAPH and TEXT, try entering GRAPH then LIST. Then enter TEXT and LIST again.

Another way to clear the screen of text is to press <CTRL/L> (hold down the CTRL key and type L) while BASIC is expecting a command (i.e. after the "Ready:" prompt).

During all of this the diagonal line has remained unchanged on the screen. This is because it is being produced from the high resolution graphics memory; this is entirely separate from the VDU memory from which text and low resolution graphics are displayed. One of the functions of the call to RESOLUTION in line 20 is to clear the high resolution memory. Try typing:

```
CALL "RESOLUTION",0,2
```

(no line number). The line disappears. Note that you have just made a CALL to a graphics routine in "direct" mode (without a line number), as opposed to including the

call in a stored program. This is often useful when experimenting.

SETTING THE RESOLUTION

The resolution of a graphics display is determined by the number of separate picture elements used to create an image. In low resolution graphics, the number of picture elements is small but the size of each is large. In high resolution graphics, greater image clarity and definition is obtained by increasing the number of picture elements and reducing their size. A picture element is ofter referred to as a pixel.

Another function of the RESOLUTION routine is (as suggested by its name) to set the resolution of the high resolution graphics display. Three modes are available: "high resolution" mode (HR) in which the display resolution is 320 pixels horizontally by 192 pixels vertically; "medium resolution" (MR) with a resolution of 160 pixels horizontally by 96 pixels vertically; and "extra high resolution" mode (EHR) with a resolution of 640 pixels horizontally by 192 pixels vertically.

A call to RESOLUTION is of the form:

```
CALL "RESOLUTION",R,B
```

where R is a number or numeric expression that can take the value 0, 1 or 2 and determines whether the resolution will be high, medium or extra high respectively, and B sets the amount of memory used by each picture element (see below). Thus the call to RESOLUTION in the example program had "argument" R equal to zero and therefore set high resolution mode. Try changing line 20 to:

```
20 CALL "RESOLUTION",1,4
```

and then running the program again. As can be seen, the diagonal line is twice as long as we are now in medium resolution mode; it is also rather dim as its intensity is only 3 compared to a maximum of 15.

The second argument of the call to the RESOLUTION routine sets the number of bits to be used for each pixel. In HR mode, two bits are available for each pixel; this gives a choice of 4 intensities (colours or shades of grey) since the two bits can represent values of 0, 1, 2, or 3. With MR, four bits are available allowing you to choose from 16 intensities, while in EHR mode only one bit is available, giving a choice of just two intensities (black or white). The effect of restricting the graphics storage for each pixel to two bits for HR mode or four bits for MR mode will be described later.

A call to the RESOLUTION routine should always be made before starting a picture and it should be thought of as initializing the graphics system. Failure to supply both arguments will result in the error message:

```
Bad no of args
```

PLOTTING POINTS AND LINES

Lines 30 and 40 of the example program are responsible for drawing the diagonal line. PLOT causes a dot to be plotted; its general form is:

```
CALL "PLOT",X,Y,I
```

which plots a dot of intensity I at position (X,Y). The origin, position (0,0), is set by RESOLUTION to the bottom left corner of the graphics area. In HR mode, the other three corners are, going clockwise, (0,191), (319,191) and (319,0). On a 380Z, however, the far right column (319) produces dimmer intensities. Therefore we shall assume 318 to be the final column. Change line 20 of the program back to:

```
20 CALL "RESOLUTION",0,2
```

then change line 40 onwards to:

```
40 CALL "PLOT",0,191,1
50 CALL "PLOT",318,191,2
60 CALL "PLOT",318,0,3
```

then RUN. The program puts a dot in each corner of the screen. The bottom left dot is bright (intensity 3), as is the bottom right one. The top left one should be rather dim (intensity 1) and the top right one somewhat brighter. If you cannot see all four dots, try adjusting the brightness and contrast controls of your TV/monitor.

Like PLOT, the call to LINE is of the general form:

```
CALL "LINE",X,Y,I
```

but now (X,Y) specifies the end point of a line. The start point is given either by a call to PLOT, or by the end point of a call to LINE, whichever occurred most recently. (RESOLUTION initializes these "memorised" XY coordinates to (0,0).) To draw a rectangular box enclosing the screen, change PLOT to LINE in lines 40 to 60:

```
40 CALL "LINE",0,191,1
50 CALL "LINE,318,191,2
60 CALL "LINE",318,0,3
```

and add:

```
70 CALL "LINE",0,0
```

and RUN.

Note that the omission of the intensity argument in line 70 is deliberate. It can be left out in calls to both PLOT and LINE; the most recently supplied value is used, in this case intensity 3. A call to RESOLUTION resets the "memorised" intensity to zero.

There are two important reservations which should be borne in mind when using the call to LINE. Firstly, the LINE call does NOT plot the first point of the line, which is assumed to have been output by a preceding call to PLOT or LINE. Secondly, in

general, a line drawn in one direction will NOT be identical to a line joining the same two points but drawn in the opposite direction. This arises as a result of the high speed line drawing algorithm used. If you later wish to erase a line, make sure you draw over it in the same direction as it was originally drawn.

RANGE OF XY COORDINATES

As already mentioned, the visible screen extends in HR from (0,0) to (319,191). The values of X and Y passed in calls to PLOT and LINE are not restricted to this range, however. In fact X and Y can specify any point on a "virtual" screen which extends from —32768 to +32767 in both axes. You can demonstrate this by adding:

```
80 CALL "PLOT",1000,1000,3
90 CALL "LINE",-1000,-1000
```

then typing RUN. A line will be drawn through the origin at intensity 3.

If the X or Y value (or indeed any value in a CALL) is greater than 65535 or less than —65536 the error message:

```
Illegal function
```

appears. Values between +32767 and +65535 are treated by PLOT and LINE as though 65536 had been subtracted first, and similarly, values from —65536 to —32769 have 65536 added. To avoid confusion you are advised only to use values in the range —32767 to +32767.

MOVING THE ORIGIN

In order to simplify graphics programs the OFFSET call allows the visible screen to be moved with respect to the virtual screen by setting the XY coordinates of the lower left corner of the visible screen. Its general form is:

```
CALL "OFFSET",X,Y
```

If, for example, the statement:

```
25 CALL "OFFSET",-50,-50
```

is added to the program, the lines that met at (0,0) move 50 screen units upwards and to the right. Try this first, then try:

```
25 CALL "OFFSET",-1050,-1050
```

You can now see the end point of the long line drawn by line 90 although none of the rectangle is visible.

Note that OFFSET does not move any graphics already displayed; it merely resets the visible screen for graphics drawn subsequently. You can verify this by deleting line 25

and typing the same statement at line 75. Both parts of the picture are now seen. Note also that RESOLUTION always resets the offset to (0,0).

OFFSET is sometimes convenient when plotting graphs. Try the following program:

```
10 GRAPH : TEXT
20 CALL "RESOLUTION",0,2
30 CALL "OFFSET",0,-96
40 LET A=16*ATN(1)/319
50 FOR X=0 TO 319
60 LET Y=50*SIN(A*X)
70 CALL "PLOT",X,Y,3
80 NEXT X
```

which plots a sine wave across the middle of the screen. In this case OFFSET avoided the need to add a value to the Y coordinate calculated at line 60.

If you are ready for a rest, you might like now to have a look at the STAR program (listed at the end of this chapter) which illustrates the use of the calls so far described.

BLOCK FILL

The FILL call provides a means for the rapid filling of a rectangular area of the screen, for example for drawing histograms. Its general form is:

```
CALL "FILL",X1,Y1,X2,Y2,I
```

This fills in the area whose lower left and top right corners are specified by the points (X1,Y1) and (X2,Y2), respectively. To draw a filled rectangle X2 should be greater than X1 and Y2 should be greater than Y1. If either X1 and X2 or Y1 and Y2 are the same then a vertical or horizontal line is drawn. If both X1,X2 and Y1,Y2 are the same then a dot is drawn. If X2 is less than X1 or if Y2 is less than Y1 then nothing is drawn.

As with PLOT and LINE, the intensity argument I can be omitted; if so the most recently used value applies

See the examples at the end of this chapter (COSINE HISTOGRAM) for an example of block fill.

THE INTENSITY ARGUMENT

In the examples so far, the intensity argument I has ranged from 0 to 3 in high resolution, from 0 to 15 in medium resolution or takes either of the values 0 or 1 in extra high resolution, where 0 corresponds to the background (initially black) and the non-zero values to progressively lighter shades of grey. In this mode plotted data overwrites any pre-existing information. For example:

```
10 CALL "RESOLUTION",0,2
20 CALL "FILL",50,50,150,150,2
30 CALL "PLOT",0,0,3
40 CALL "LINE",200,200
50 A$=GET$()
60 CALL "PLOT",0,0,0
70 CALL "LINE",200,200
```

This program first draws a grey square (red on a colour monitor) then overwrites it with a diagonal white line. Line 50 merely causes a pause until a key is pressed. The white line is erased by replotting it at intensity zero, leaving however a "black" line through the grey square.

This can be avoided by using EXCLUSIVE-OR (XOR) plotting. Considering two binary digits (bits) A and B, the truth table of the XOR function is:

A	B	A XOR B
0	0	0
1	0	1
0	1	1
1	1	0

If we consider A to be the intensity bit we are plotting and B to be the existing screen content, it can be seen that when plotting a one, if the corresponding bit of the screen is off, it is turned on (set to one), while if it is already on it is turned off (set to zero). The important feature of XOR plotting occurs when a point is replotted in XOR mode at the same intensity. This is equivalent to combining A with A XOR B in the above table. It can be seen that the new result is identical with B. In other words, a point plotted in XOR mode is reversible without loss of the original information.

XOR plotting takes place when the intensity argument is negative. In high resolution, values for I of −1 to −3 correspond to values of +1 to +3 but are plotted by exclusive-OR rather than by replacement.

To demonstrate this, change lines 30 and 60 to:

```
30 CALL "PLOT",0,0,-3
60 CALL "PLOT",0,0,-3
```

Now, when the original line is replotted, the square is left intact.

XOR plotting is useful for drawing data provisionally (e.g. an "elastic band" line from a fixed point to a roving cursor) or for temporarily highlighting an area by means of an XOR block fill. Note that both intensity settings have the same (negative) value for I, as opposed to replacement plotting when the two (positive) values are different.

The example program STRIPES at the end of this chapter uses XOR plotting.

PEN UP MOVEMENTS

It is occasionally convenient to reset the memorised (X,Y) coordinates (the start point of a line) without causing any graphical output. This occurs when an intensity argument of 16 is used.

MODIFYING THE DISPLAYED INTENSITY

A powerful feature of the Research Machines high resolution graphics in HR and MR modes is the ability to change almost instantaneously the displayed brightness that corresponds to one of the plotted intensities. For example, intensity 3 could first be set to display at the same level as the background. A complex picture could then be drawn, perhaps taking several seconds. Finally intensity 3 could be set to display as white, causing the complete picture to appear in an instant.

In HR and MR modes, the mapping of the plotted intensity (in the range 0 to 3 or 0 to 15 as previously described) to the displayed intensity, a value from 0 to 255, corresponding to shades of grey from black to white, is carried out by means of a special hardware area on the graphics board called the colour lookup table and the values this contains are set by the COLOUR call. In EHR mode the mapping of the intensity value does not pass through the colour lookup table and consequently only black (0) or white (1) are available.

In HR and MR modes, a call to RESOLUTION sets up default values in the colour lookup table. For example, in high resolution, the values assigned to plotted intensities of 0, 1, 2 and 3 are 0, 64, 128 and 255, corresponding to black, dark grey, light grey and white. In medium resolution, the intensity arguments 0 to 15 are similarly mapped to black and 15 progressively lighter shades of grey, again arranged in an approximately logarithmic series, which the eye sees as roughly equal steps of brightness.

These default values can be changed at any time by a call to COLOUR, in HR and MR modes. In EHR mode, a call to COLOUR will have no effect. The call is of the general form:

```
CALL "COLOUR",I,N
```

where I is the plotted intensity value (range 0 to 3 for high resolution, 0 to 15 for medium) and N is the new displayed intensity (range 0 to 255). For example, in high resolution, the statements:

```
100 CALL "COLOUR",0,255
110 CALL "COLOUR",1,128
120 CALL "COLOUR",2,64
130 CALL "COLOUR",3,0
```

would reverse any graphics currently displayed, causing the appearance of a "negative". Note that a call to COLOUR can easily be placed in a loop, allowing a picture to be faded up or down. For example:

```
140 FOR N=1 TO 255
150 CALL "COLOUR",3,N
160 NEXT N
```

fades intensity 3 up from 0 (black) to 255 (white) again.

A call to COLOUR with no arguments:

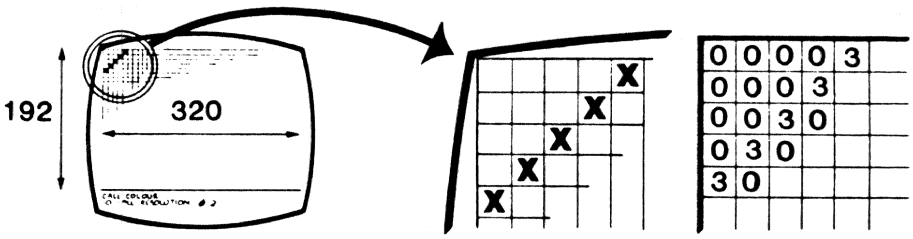
```
170  ALL "COLOUR"
```

restores the default values as set by RESOLUTION.

For example type in:

```
CALL "RESOLUTION",0,2
CALL "PLOT",0,187,3
CALL "LINE",4,191,3
```

and a small line will be drawn in the top left hand corner of the screen. The situation will be as shown below:



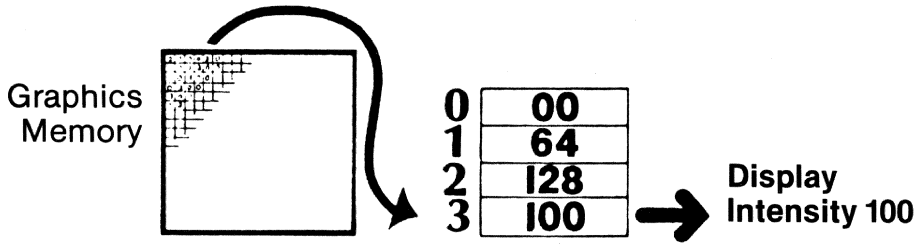
Screen Display *Graphics Memory*

Each pixel that makes up the line holds the value 3, but the displayed intensity for each pixel is 255.

If you now type

```
CALL "COLOUR",3,100
```

the value 100 will be placed into position of 3 of the colour lookup table replacing value 255 and so the colour of the whole line will change from 255 to 100, as shown overleaf.



Colour Lookup Table

SETTING THE COLOUR

In the description so far, it has been assumed that the graphics output is being displayed in black and white. The graphics memory can also display in colour, provided a colour monitor is available. As described in the previous section, the hardware colour lookup table maps the plotted intensity (pixel information) to a value between 0 and 255, that is, to an 8-bit value. When using a colour monitor it is often more convenient to use an alternative form of the colour call:

```
CALL "COLOUR",I,R,G,B
```

where I corresponds to the plotted intensity as before and R, G and B are the required values for red, green and blue. Parameters R and G may take values between 0 and 7. On a 480Z, values of 4,5,6 or 7 turn the red or green colour on, whereas values 0,1,2 or 3 turn the red or green colour off. On a 380Z, the value specifies the colour intensity (0 for colour off) with 7 representing the highest intensity. Parameter B may take any value between 0 and 3. On a 480Z a value of 2 or 3 will turn the blue colour on and 0 or 1 will turn the blue colour off. On a 380Z the vlaue of parameter B specifies the blue colour intensity. Giving R, G and B values of 0 will generate black and giving R and G values of 7 and B a value of 3 will generate white. On a 480Z system, combinations of red, blue and green will generate the three colours yellow, magenta or cyan.

PROGRAMMING MORE THAN ONE PICTURE

So far the discussion has been confined to a single picture. In high resolution with two bits/pixel and extra high resolution with one bit/pixel only one picture is available; the whole of the usable graphics memory is needed. In the other mode mentioned, medium

medium resolution with four bits/pixel, only half of the graphics memory is used and two separate pictures can be constructed.

These are considered to occupy two "pages" of the (medium resolution) graphics memory and the pages are numbered 0 and 1. It is important to note that you can display either page 0 or page 1, but never both together. (This is a consequence of the way the graphics hardware is designed.) After a call to RESOLUTION, page 0 is selected both for display and for update. Calls to PLOT, LINE and FILL write to page 0 and the display circuits show the contents of page 0 on the screen.

The page on which plotting takes place can be changed by a call to UPDATE of the general form:

```
CALL "UPDATE",P
```

where P has the value 0 or 1, and similarly, the page that is displayed can be changed by a call to DISPLAY of the form:

```
CALL "DISPLAY",P
```

Notice that the separation of the update and display functions allows you to display one picture while generating another, thus allowing a degree of animation without the relatively slow process of picture generation being visible.

The following simple example clarifies this:

```
 10 GRAPH : TEXT
 20 CALL "RESOLUTION",1,4
 30 CALL "UPDATE",0
 40 CALL "PLOT",0,0,8
 50 CALL "LINE",159,95
 60 CALL "UPDATE",1
 70 CALL "PLOT",0,95,15
 80 CALL "LINE",159,0
 90 INPUT "Display which page";P
100 CALL "DISPLAY",P
110 GOTO 90
```

Page 0 displays a medium grey line from bottom left to top right, page 1 a white line from top left to bottom right. Line 90 allows you to select one or the other. Type CTRL/Z to exit.

You are reminded that these two pages are only available in medium resolution.

USING A REDUCED NUMBER OF BITS/PIXEL

The graphics software also supports three additional modes in which the number of bits/pixel is reduced. The effect of this is to restrict the number of separate intensities that can be displayed and instead to allow multiple pictures to be generated. In a way the result is similar to the concept of "pages" in medium resolution which has just

been introduced but since the effect is available in high resolution as well and since the mechanism is different, the name "view" has been chosen. Considering high resolution, we can either select 2 bits/pixel (as we have done so far) or 1 bit/pixel and have two views. Multiple views are not possible in extra high resolution since only 1 bit/pixel is available for the intensity information. Like pages, views are normally displayed separately, but since view selection is carried out by software manipulation of the colour lookup table, it is possible to arrange to display more than one view at once.

As indicated earlier, the choice of number of bits/pixel is made by the arguments used in a call to RESOLUTION. Thus the call:

```
10 CALL "RESOLUTION",0,1
```

sets high resolution with one bit/pixel and two views, view 0 and view 1. The choice of view is made in the same way as for page in medium resolution; thus CALL "UPDATE",1 selects view 1 for PLOT, LINE and FILL; CALL "DISPLAY",1 results in view 1 being displayed. As before, RESOLUTION automatically selects view 0 both for update and display. Note that in this mode, the intensity argument I can only take on values of 0 or 1. (In fact the I argument is ANDed with the maximum value allowed, so that if, for example, 2 is supplied, 2 AND 1 (= 0) is used for plotting.) However as before the shades of grey or colours that correspond to the two intensities can be changed by calls to COLOUR. For example:

```
10 CALL "RESOLUTION",0,1
20 CALL "COLOUR",0,0,0,3
30 CALL "COLOUR",1,7,0,0
40 CALL "PLOT",0,0,1
50 CALL "LINE",319,191
```

will plot a red diagonal line on a blue background as view 0.

MULTIPLE VIEWS IN MEDIUM RESOLUTION

The remaining two modes that can be selected are both in medium resolution. If the pixel is restricted to one bit by a call of the form:

```
10 CALL "RESOLUTION",1,1
```

we still have two pages but each contains four views, making eight pictures in all. The intensity argument may take on values of 0 or 1. Calls to UPDATE and DISPLAY require two arguments and are of the form:

```
CALL "UPDATE",P,V
CALL "DISPLAY",P,V
```

where P is the page (range 0 to 1) and V is the view (range 0 to 3). The N argument in a call to COLOUR must lie in the range 0 to 1.

Similarly, a two bit pixel can be selected, giving a choice of two pages, each with two

views and four intensities.

The six possible modes are conveniently referred to as HR2 (high resolution, 2 bits/pixel), EHR, HR1, MR4, MR2 and MR1.

Notice that although UPDATE and DISPLAY accept a variable number of arguments (page, view or both, depending on mode) both page and view can always be supplied without confusion. In MR4 for example, the view argument is ignored if supplied, while in HR1 the page argument is discarded.

The program REVOLVE, in the examples to be found at the end of this chapter, shows how eight medium resolution pictures can create the illusion of motion when displayed in rapid succession.

CLEARING A VIEW

When working with more than one view it is frequently desirable to clear a view prior to updating it. (This will usually be done invisibly in the "background" by selecting another completed view for display.) One means of doing this, having selected the view for update, is simply to block fill it with intensity 0. For example, in HR1, the statements:

```
50 CALL UPDATE,1
60 CALL FILL,0,0,319,191,0
```

would clear view 1. As a convenience, the call to CLEAR has the same effect. Thus the above example can be replaced by:

```
50 CALL UPDATE,1
60 CALL CLEAR
```

CLEAR absolves the programmer from having to be aware (at that moment) of the current offset or resolution.

DISPLAYING MORE THAN ONE VIEW

This Section can be skipped on first reading; it is only necessary to study it when you wish to display more than one view at once.

Multiple views (but NOT multiple pages) are achieved by manipulation of the colour lookup table. Consider high resolution; normally, with 2 bits/pixel, intensities 0 to 3 are arranged to map to black, dark grey, light grey and white in the colour lookup table (CLT). We may represent this as:

(a)

PIXEL	CLT
00	0
01	64
10	128
11	255

If instead we arrange that only pixels in which the less significant bit are set are displayed at an intensity different from the background, these become the only pixels that are visible. The colour lookup table might become:

(b)

PIXEL	CLT
00	0
01	200
10	0
11	200

Information represented by the more significant bit is not differentiated from background (except where the less significant bit is also set). Alternatively we can arrange to suppress the information carried in the less significant bit:

(c)

PIXEL	CLT
00	0
01	0
10	200
11	200

and only see the more significant bit.

This is the means used to achieve the separate views. After a call to RESOLUTION which specifies a restricted number of bits/pixel (modes HR1, MR1 and MR2), DISPLAY swaps around the colour lookup table in the manner outlined above, UPDATE arranges that PLOT, LINE, FILL and CLEAR only modify the appropriate part of each pixel and COLOUR modifies the entries in the colour lookup table appropriate to the currently selected view. Thus in the illustrations of colour table organisation above, intensity 0 corresponds to a displayed intensity of zero and intensity 1 to 200, but the positions in which 0 and 200 appear depend on whether view 0 or view 1 is selected for display.

It is by manipulation of these positions that the programmer can select a combination of views. Thus to display both view 0 and view 1, the colour table is changed to:

(d)

PIXEL	CLT
00	0
01	200
10	200
11	200

while to display only those pixels which are common to view 0 and to view 1, the table is:

(e)

PIXEL	CLT
00	0
01	0
10	0
11	200

It should be realised that case (a) is automatically selected when the resolution is 2 bits/pixel, and that with 1 bit/pixel, cases (b) and (c) occur automatically as a result of a call to DISPLAY requesting view 0 or 1. It is only when case (d) or case (e) is required that programmer intervention is necessary.

Two additional calls allow direct manipulation of the colour table. SETCOL sets a specified element of the colour table to a display intensity value. Its general form is the same as that for COLOUR, namely:

```
CALL "SETCOL",I,N
```

where I is the "address" in the colour table to modify and N is the new value. Again, an alternate form is available for use with a colour monitor:

```
CALL "SETCOL",I,R,G,B
```

Calls to SETCOL do not actually effect any change in the contents of the colour table but merely set up the values in memory (in a special copy of the colour table which is maintained in all modes). When all the values to be changed have been set up, the colour table is loaded by a call to VIEW of the form:

```
CALL "VIEW",P
```

The P argument indicates which medium resolution page is to be displayed. It can be left out in high resolution but must be supplied in medium.

When using SETCOL and VIEW it is important for the programmer to realise that he is taking over functions that are normally carried out automatically. Thus in HR modes he must set up four intensity entries with SETCOL even when, as in mode HR1, there are only two logical intensities in use. Similarly, in the MR modes, all sixteen intensity entries must be defined. (The logical intensities are normally multiplexed into the colour lookup table by DISPLAY or COLOUR.)

The following program selects mode HR1, setting up case (d) with view 0 and view 1 displayed simultaneously:

```
10 CALL "RESOLUTION",0,1
20 DATA 0,200,200,200
30 FOR I=0 TO 3
40 READ N
50 CALL "SETCOL",I,N
60 NEXT I
70 CALL "VIEW"
```

You need only change line 20 to:

```
20 DATA 0,0,0,200
```

to set up case (e), displaying instead those pixels that are common to view 0 and view 1.

Note that this method is also of general application in all modes (except EHR). It provides a means of deferring the alteration of the colour table until all the colours have been selected. This is not normally of concern in high resolution, but in medium resolution, the changing of all 16 colours by use of the COLOUR call does cause a discernible "ripple" owing to the finite time a call to COLOUR takes. (This is partially because COLOUR waits for the end of a frame, up to 20 ms, before changing the colour table. Sixteen calls to COLOUR therefore take at least 320 ms.) Instead, then, the sixteen colours can be selected by SETCOL and finally loaded into the table by VIEW, thus changing all the displayed colours at once. Examples of both methods follow:

```
10 CALL "RESOLUTION",1,4
20 DATA ...specifying the colours
30 FOR I=0 TO 15
40 READ R,G,B
50 CALL "COLOUR",I,R,G,B
60 NEXT I
```

```
10 CALL "RESOLUTION",1,4
20 DATA ...specifying the colours
30 FOR I=0 TO 15
40 READ R,G,B
50 CALL "SETCOL",I,R,G,B
60 NEXT I
70 CALL "VIEW",0
```

Remember that VIEW must be followed by page in medium resolution.

User alteration of the colour table is more complicated in modes MR1 and MR2 than in mode HR1 but follows the same general principles. Appendix E contains tables showing which colour table elements affect which view.

The example program GRAPH illustrates the use of SETCOL and VIEW in displaying two views simultaneously.

EXAMPLES

This section contains a number of BASIC programs that generate pictures. Their main purpose is to illustrate various aspects of the graphics support routines; they are not intended to perform useful functions. However they do embody a few sections which may be of use in other programs.

STAR

This program illustrates line drawing in high resolution. The pictures are drawn with 2 bits/pixel but since only intensity 3 is used, they could equally well be in mode HR1. OFFSET is used to simplify the calls to PLOT and LINE.

```
100 REM-- STAR
110 DIM X(15),Y(15)
120 R1=110: REM X RADIUS
130 R2=90: REM Y RADIUS
140 X0=159: REM X CENTRE
150 Y0=95: REM Y CENTRE
160 GRAPH : TEXT
170 P2=ATN(1)*8
180 Z=3
190 FOR N=3 TO 15 :REM for each start
200 A1=P2/N
210 FOR J=1 TO N :REM compute coordinates
220 A=A1*J
230 X(J)=R1*COS(A)
240 Y(J)=R2*SIN(A)
250 NEXT J
255 REM Draw Star
260 CALL "RESOLUTION",0,2
270 CALL "OFFSET",-X0,-Y0
280 FOR J=1 TO N-1
290 FOR K=1 TO N-J
300 CALL "PLOT",X(K),Y(K),Z
310 CALL "LINE",X(K+J),Y(K+J)
320 NEXT K
330 NEXT J
340 REM-- WAIT A WHILE
350 FOR J=0 TO 1000: NEXT J
360 NEXT N
370 GOTO 190
```

COSINE HISTOGRAM

This program shows how to draw a histogram of data which in real life might be the frequency distribution of events, etc. The histogram is drawn in two tones to make it more dramatic (remove line 180 to keep to one tone).

Lines 240 onwards are a simple general-purpose axis routine where XL and XH are the absolute X coordinates of the ends of the X axis, XY is its Y coordinate, XI is the X axis increment and TI is the size of the tick marks. YL, YH, YX and YI are the corresponding Y axis values.

```
100 REM-- COSINE   HISTOGRAM
110 GRAPH    :TEXT
120 CALL "RESOLUTION",0,2
130 A=8*ATN(1)/320
140 Z=0

150 FOR   J=0 TO 310 STEP 10
160 Y=80*(1-COS(A*(J+5)))
170 CALL "FILL",J,0,J+9,Y,Z+1
180 Z=1-Z
190 NEXT J

200 XL=0:XH=318:XY=0:XI=10
210 YL=0:YH=191:YX=160:YI=10:TI=2
220 GOSUB 240
230 END

240 REM-- DRAW AXES
250 CALL "PLOT",XL,XY,3
260 CALL "LINE",XH,XY
270 FOR X=XL TO XH STEP XI
280 CALL "PLOT",X,XY-TI,3
290 CALL "LINE",X,XY+TI
300 NEXT X
310 CALL "PLOT",YX,YL,3
320 CALL "LINE",YX,YH
330 FOR Y=YL TO YH STEP YI
340 CALL "PLOT",YX-TI,Y,3
350 CALL "LINE",YX+TI,Y
360 NEXT Y
370 RETURN
```

STRIPES

This short program draws (cubist?) pictures. It demonstrates the speed of the block fill and provides a trivial example of "XOR plotting" (variable C may be negative). A more important example of the use of XOR plotting is to remove a point or a line from the display. This is done by replotting the same data using negative intensities.

```
100 REM-- STRIPES
110 GRAPH : TEXT
120 RANDOMIZE
130 CALL "RESOLUTION",1,4
140 X0=RND(1)*160
150 X1=RND(1)*160
160 IF X0>X1 THEN T=X0: X0=X1: X1=T
170 Y0=RND(1)*96
180 Y1=RND(1)*96
190 IF Y0>Y1 THEN T=Y0: Y0=Y1: Y1=T
200 C=INT(RND(1)*31-15)
210 CALL "FILL",X0,Y0,X1,Y1,C
220 GOTO 140
```

REVOLVE

This program shows how you can draw up to 8 pictures in separate pages and views (selected by variables P and V), then display them in sequence to give an impression of movement. In order to make the program as clear as possible, the pictures have been kept very simple.

```
100 REM-- REVOLUTION
110 CALL "RESOLUTION",1,1
120 GRAPH :TEXT
130 REM-- CREATE 8 PICTURES
140 PI=ATN(1)*4
150 XO=80
160 YO=42
170 R=40
180 A=PI/8
190 FOR P=0 TO 1
200 FOR V=0 TO 3
210 CALL "UPDATE",P,V
220 A1=A*(4*P+V)
230 X=R*COS(A1)+XO
240 Y=R*SIN(A1)+YO
250 CALL "PLOT",X,Y,1
260 X=R*COS(A1+PI)+XO
270 Y=R*SIN(A1+PI)+YO
280 CALL "LINE",X,Y,1
290 CALL "DISPLAY",P,V
300 NEXT V
310 NEXT P
320 REM-- DISPLAY EACH IN TURN
330 FOR P=0 TO 1
340 FOR V=0 TO 3
350 CALL "DISPLAY",P,V
355 REM-- WAIT A WHILE
360 FOR I=1 TO 10
370 NEXT I
380 NEXT V
390 NEXT P
400 GOTO 330
```

GRAPH

This program plots a series of graphs of random data on a grey grid. Block fill is used to plot a little square at each graph point; the points are joined by lines. High resolution with one bit/pixel is used, giving two views. The grid is plotted as view 0 and its intensity is set to 128 by line 140 (it would be 200 by default). The graphs are drawn as view 1 but the SETCOL and VIEW calls are used to load the colour lookup table directly, thus allowing view 0 and view 1 to be displayed simultaneously. Intensities 2 and 3 govern the intensity of the graph; this is set to 255 by lines 260-280. At line 440 the graph is made invisible by setting intensity 2 to 0 and intensity 3 to the grey grid colour (to deal with graph points on the grid). Then the graph is erased by calling CLEAR. Note that the grid remains visible throughout.

This is a fairly advanced example illustrating direct loading of the colour lookup table.

```
100 REM-- GRAPH DRAWING
110 GRAPH :TEXT
120 CALL "RESOLUTION",0,1

130 REM-- MAKE GRID
140 CALL "COLOUR",1,128
150 FOR X=0 TO 310 STEP 10
160 CALL "PLOT",X,0,1
170 CALL "LINE",X,190
180 NEXT X
190 FOR Y=0 TO 191 STEP 10
200 CALL "PLOT",0,Y,1
210 CALL "LINE",310,Y
220 NEXT Y
230 CALL "UPDATE",1
240 REM-- LOOP TO PLOT GRAPHS
250 REM-- DISPLAY VIEW 0 AND VIEW 1
260 DATA 0,128,255,255
270 RESTORE 260
280 GOSUB 510
290 I=20:J=2
300 REM-- DO 1ST POINT
310 X0=0:Y0=RND(1)190
320 CALL"FILL",X0-J,Y0-J,X0+J,Y0+J,1
330 REM-- DO REST OF GRAPH
340 FOR X=I TO 310 STEP I
350 Y=RND(1)*190
360 CALL"PLOT",X0,Y0,1
370 CALL"LINE",X,Y
380 CALL"FILL",X-J,Y-J,X+J,Y+J
390 X0=X:Y0=Y
400 NEXT X

410 REM-- WAIT A WHILE
420 FOR K=0 TO 4000
```

```
430 NEXT K
440 REM-- BLANK  VIEW 1
450 DATA 0,128,0,128
460 RESTORE 450
470 GOSUB 510

480 REM-- CLEAR VIEW 1 AND LOOP BACK
490 CALL "CLEAR"
500 GOTO 270

510 REM-- SET COLOUR VECTOR FROM DATA
520 FOR K¤0 TO 3
530 READ N
540 CALL "SETCOL",K,N
550 NEXT
560 CALL "VIEW"
570 RETURN
```

Try changing line 450 to:

```
450 DATA 0,128,0,0
```

The effect of a static grid with changing graphs can also be obtained in mode HR2 by plotting the graph in XOR mode but the graph coordinates would have to be stored to allow them to be used again for "unplotting". Try it!

CHAPTER 16

HRG ROUTINES - REFERENCE SECTION

This chapter contains the formal definitions of the graphics routines that can be called from BASIC. These routines are available from Level 1 and Level 2 graphics. For a description of extra graphics routines for Level 2 only see chapter 17. The descriptions are in alphabetical order and are set out as follows:

Form: shows the correct form for calling the routine.

Purpose: a brief summary of what the routine is used for.

Remarks: describe in detail how to use the routine.

Examples: contain sample program fragments that demonstrate how to call the routines.

NOTE 1. References to Extra High Resolution Graphics (EHR) apply only to the 480Z.

NOTE 2. Users of previous versions of BASIC will note that the method of saving pictures on (and restoring pictures from) disc has changed: HRG routines GSAVE, WRITE, READ and GLOAD are obselete, and have been superseded by GWRITE and GREAD. The old routines are mentioned here only because there may exist some programs that use them.

Routines GLOAD and GSAVE now have no effect and return without any action. Routines READ and WRITE will have their three trailing parameters ignored, otherwise be treated as GREAD and GSAVE. Thus old programs will still work as before, provided they transferred whole pictures and started from record 0 in the file.

CLEAR

Forms: `CALL "CLEAR"`

Purpose: To clear the currently selected page and view.

Remarks: CLEAR clears the page and view selected by UPDATE to logical intensity 0. It works by making an internal call to FILL and absolves the programmer from having to be aware of the current resolution and offset. It is useful when clearing selected views but calling RESOLUTION is faster when it doesn't matter that the whole of graphics memory is cleared.

Example:
```
10 CALL "RESOLUTION",1,1
20 CALL "UPDATE",1,3
30 CALL "CLEAR"
```

clears view 3 of page 1.

COLOUR

Forms:

```
CALL "COLOUR",I,N
CALL "COLOUR",I,R,G,B
CALL "COLOUR"
```

Purpose: To set the actual intensity or colour displayed on the screen that corresponds to a logical intensity I.

Remarks: COLOUR sets the value in the colour lookup table that corresponds to the logical intensity I plotted by PLOT, LINE or FILL. In EHR mode this CALL will have no effect.

The first form with two arguments is used with black and white displays. The value of N can range from 0, which displays as black, to 255, displaying as white, with a continuous range of progressively lighter shades of grey in between.

The second form with four arguments is for colour displays. On a 480Z, values of 4,5,6 or 7 turn the red or green colour on, whereas values 0,1,2 or 3 turn the red or green colour off. On a 380Z, the value specifies the colour intensity (0 for colour off) with 7 representing the highest intensity. Parameter B may take any value between 0 and 3. On a 480Z a value of 2 or 3 will turn the blue colour on and 0 or 1 will turn the blue colour off. On a 380Z the value of parameter B specifies the blue colour intensity. Giving R, G and B values of 0 will generate black and giving R and G values of 7 and B a value of 3 will generate white. On a 480Z system, combinations of red, blue and green will generate the three colours yellow, magenta or cyan.

Calling RESOLUTION sets up default values in the colour lookup table such that increasing logical intensity corresponds to approximately equal steps of increasing brightness. (These default values are in general only suitable for black and white displays.)

The third form of the call to COLOUR with no arguments restores the default values set by RESOLUTION.

The table below shows the values of R, G and B which will generate each of the eight possible colours on a 480Z.

R	G	B	COLOUR
<4	<4	<2	Black
<4	<4	>=2	Blue
<4	>=4	<2	Green
>=4	<4	<2	Red
>=4	>=4	<2	Yellow
>=4	<4	>=2	Magenta
<4	>=4	>=2	Cyan
>=4	>=4	>=2	White

Example:

```
10 CALL "RESOLUTION",0,2
20 DATA 255,128,64,0
30 FOR I=0 TO 3
40 READ N
50 CALL "COLOUR",I,N
60 NEXT I
```

sets the colour lookup table to produce a reversed image. (The default values for the lookup table in mode HR2 are 0, 64, 128 and 255.)

DISPLAY

Forms:

```
CALL "DISPLAY",P,V      (all modes)
CALL "DISPLAY",V        (HR1)
CALL "DISPLAY",P        (MR4)
```

Purpose: To display page P, view V.

Remarks: In mode HR2 and EHR there is only one page and view and this call is not used. In all MR modes there are two separate pages, numbered 0 and 1. In modes HR1 and MR2 there are two views, numbered 0 and 1, and in mode MR1 there are four views, numbered 0 to 3.

After a call to RESOLUTION, page 0 view 0 is selected for display. Calling DISPLAY changes the page and/or view of graphics memory that is displayed. Similarly, calling UPDATE selects the page and view that will be modified by PLOT, LINE and FILL.

P may be omitted in mode HR1 and V in mode MR4.

Example:

```
10 CALL "RESOLUTION,1,1
20 CALL "DISPLAY",1,3
```

set mode MR1, then select page 1, view 3 for display.

FILL

Forms:
```
CALL "FILL",X1,Y1,X2,Y2,I
CALL "FILL",X1,Y1,X2,Y2
```

Purpose: To fill the rectangle whose lower left corner is (X1,Y1) and upper right corner is (X2,Y2) with intensity I.

Remarks: The point (X1,Y1) specifies the lower left corner and the point (X2,Y2) the upper right corner of a rectangle to be filled in. These XY values can lie in the range -32768 to +32767 but extend only to the edge of the screen. Thus X1 can never be less than the left hand border, X2 never more than the right border, Y1 never less than the lower border and Y2 never more than the upper border. If X1 and X2 are the same but Y2 is greater than Y1, a vertical line is drawn, and if Y1 and Y2 are the same but X2 is greater than X1, a horizontal line is drawn. If X1 is the same as X2 and Y1 is the same as Y2, a point is plotted. If X2 is less than X1 or if Y2 is less than Y1 then there is no output.

I determines the logical intensity of the block in the same way as for PLOT (see elsewhere in this chapter). If omitted the value last specified in a call to PLOT, LINE or FILL is used.

The "memorised" XY coordinates are not affected by FILL.

Example:
```
10 CALL "FILL",50,50,100,100,2
```

fill the block whose corners are (50,50) and (100,100) with intensity 2.

GREAD

Form: `CALL "GREAD",A`

Purpose: To load into graphics memory a picture stored in a disc file.

Remarks: GREAD requires one parameter, specifying the channel number by which the disc file is to be read. This number will normally be 10 (especially in Extended BASIC Version 5).

The disc file to be read should have been written by GWRITE (if it was not, see Note 2 on page 16.1).

If the program that wrote the disc file stored the contents of the colour lookup table as well as the picture elements, then the original intensities or colours can also be restored by means of some GET commands and calls to subroutine COLOUR.

If GREAD reads a file containing insufficient data to fill the entire screen, it reports the fact with the error message:

```
Insufficient data
```

Example: The following program will read and display the picture stored in a disc file by the program used as an example under GWRITE:

```
10 CALL "RESOLUTION",0,2
20 OPEN #10,"NEWFILE.PIC"
30 CALL "GREAD",10 :REM Reconstruct picture
35                  REM from disc file and
40                  REM Find colours used
50 C1=GET(#10): CALL "COLOUR",0,C1
60 C2=GET(#10): CALL "COLOUR",1,C2
70 C3=GET(#10): CALL "COLOUR",2,C3
80 C4=GET(#10): CALL "COLOUR",3,C4
90 CLOSE INPUT #10
```

GWRITE

Form: `CALL "GWRITE",A`

Purpose: To save the contents of graphics memory in a disc file.

Remarks: GWRITE requires one parameter, specifying the channel number by which the disc file is to be written. This number will normally be 10 (especially in Extended BASIC Version 5).

Disc files written by GWRITE are expected to be read by GREAD.

It may also be useful to store in the same disc file not only the picture elements, as written by subroutine GWRITE, but also the contents of the colour lookup table. This can be done by means of the PUT command, and it is recommended that, if present, this data should follow that written by GWRITE in the disc file.

Example: The following program will write to disc file NEWFILE.PIC first the picture elements, then the colour lookup table, of the current HRG screen, as required by the sample program under GREAD:

```
 10 CALL "RESOLUTION",0,2
 20 REM Yellow  Cyan    Blue   Magenta
 30     C1=237: C2=182: C3=18: C4=91
 40 CALL "COLOUR",0,C1
 50 CALL "COLOUR",1,C2
 60 CALL "COLOUR",2,C3
 70 CALL "COLOUR",3,C4
  .
  .  (Generate picture on the screen)
  .
200 REM  Save picture in file NEWFILE.PIC
210 CREATE #10,"NEWFILE.PIC
220 CALL "GWRITE",10
230 REM  Save colours used in NEWFILE.PIC
240 PUT #10,C1,C2,C3,C4
250 CLOSE #10
```

LINE

Forms:

```
CALL "LINE",X,Y,I
CALL "LINE",X,Y
```

Purpose: To draw a line on the screen of intensity I ending at the point (X,Y).

Remarks: LINE takes as its start point the end point of the previous line or the last plotted point, whichever occurred most recently, and draws a line to the end point (X,Y) specified in the call. X and Y can lie in the range −32768 to +32767; a line is only visible if it falls wholly or partially within the screen limits defined by RESOLUTION and OFFSET.

I determines the logical intensity of the line in the same way as for PLOT (see elsewhere in this chapter). If omitted the value last specified in a call to PLOT, LINE or FILL is used.

CAUTION 1: the start point of the line is NOT plotted. (If it was, exclusive-OR plotting would not work correctly.)

CAUTION 2: a line is usually not identical when drawn in the two possible directions. If you require to 'unplot' a line by replotting it with background intensity or by exclusive-OR plotting, make sure you plot it in the same direction as it was originally plotted.

Example:

```
10 CALL "PLOT",0,0,3
20 CALL "LINE",100,50
```

draws a line at intensity 3 from (0,0) to (100,50).

OFFSET

Form: `CALL "OFFSET",X,Y`

Purpose: To change the XY coordinates of the lower left corner of the screen.

Remarks: X and Y can lie between —32768 and +32767. After a call to OFFSET the lower left corner of the screen corresponds to the point (X,Y).

CAUTION: graphics already drawn prior to a call to OFFSET are unchanged.

Example:
```
10 CALL "RESOLUTION",0,2
20 CALL "OFFSET",-160,-96
```

places the origin (0,0) in the centre of the screen.

PLOT

Forms:

```
CALL "PLOT",X,Y,I
CALL "PLOT",X,Y
```

Purpose: To plot a point on the screen of intensity I at coordinates (X,Y).

Remarks: X and Y can lie in the range −32768 to +32767. However a point will only be displayed if its XY coordinates lie within the screen limits defined by RESOLUTION and OFFSET. The point (X,Y) is remembered and can be used as the start point of a line.

I determines the "logical" intensity of the point. Allowable values of I depend on the resolution and number of bits/pixel (see RESOLUTION). For example, in mode HR2 where 4 intensity values are available, the absolute value of I can range from 0 to 3.

If I is positive, the new intensity value replaces the old contents of the graphics memory at that point. If negative, the new value written to the graphics memory is the exclusive-OR of the old value and the absolute value of I. A point plotted in this way is reversible by a further call to PLOT with the same negative value of I.

If I is 16, the XY coordinates are updated but no point is displayed. An intensity of 16 corresponds to a "pen up" move on a plotter.

If I is omitted the value last specified in a call to PLOT, LINE or FILL is used. After RESOLUTION this value is zero.

Example:

```
10 CALL "PLOT",100,50,2
```

plots a point of intensity 2 at (100,50).

RESOLUTION

Form: `CALL "RESOLUTION",R,B`

Purpose: To select high, medium or extra high resolution and to set up the number of bits/pixel.

Remarks: RESOLUTION initializes the graphics hardware and clears the screen.

R may take the value 0, 1 or 2 corresponding to high, medium or extra high resolution; B sets the number of bits/pixel. The following combinations of R and B are allowed:

R	B	Mode	Pages	Views	Intensities	
0	2	HR2	1	1	4	
0	1	HR1	1	2	2	
1	4	MR4	2	1	16	
1	2	MR2	2	2	4	
1	1	MR1	2	4	2	
2	1	EHR	1	1	2	(480Z only)

where HR2 stands for "High Resolution, 2 bits/pixel" and so on, and page, view and intensity are defined at the beginning of this chapter.

After a call to RESOLUTION the lower left corner of the screen corresponds to the XY coordinates (0,0). In high resolution modes the upper right corner is (319,191); in medium resolution modes it is (159,95); in extra high resolution it is (639, 191). (These can be changed by calling OFFSET.)

RESOLUTION initializes the "memorised" XY coordinates which are used as the start point of a line to (0,0).

Example:
```
10 CALL "RESOLUTION",0,2
```

sets high resolution mode with 2 bits/pixel.

SETCOL

Forms:

```
CALL "SETCOL",I,N
CALL "SETCOL",I,R,G,B
CALL "SETCOL"
```

Purpose: To modify the values held in the memory-copy of the colour lookup table without affecting the table itself (see also VIEW and Appendix E).

Remarks: The SETCOL call sets up a value in the copy of the colour lookup table that is held in memory. This copy is transferred to the hardware table by calling VIEW. This is in contrast to the COLOUR call which both modifies the copy of the table held in memory and loads the hardware table, assuming the page and view currently selected by DISPLAY.

SETCOL thus allows the display of non-standard combinations of views (for example, in HR1, of the differences between view 0 and view 1). It also allows the loading of the colour lookup table to be deferred until all colours have been set up.

I is the logical intensity to be modified and should be between 0 and 3 in high resolution and 0 and 15 in medium resolution. In extra high resolution this CALL will have no effect.

The first form of the call with two arguments is used for black and white. N can range from 0 (black) to 255 (white). The second form with four arguments is for colour. On a 480Z, values of 4,5,6 or 7 turn the red or green colour on, whereas values 0,1,2 or 3 turn the red or green colour off. On a 380Z, the value specifies the colour intensity (0for colour off) with 7 representing the highest intensity. Parameter B may take any value between 0 and 3. On a 480Z a value of 2 or 3 will turn the blue colour on and 0 or 1 will turn the blue colour off. On a 380Z the value of parameter B specifies the blue colour intensity. Giving R, G and B values of 0 will generate black and giving R and G values of 7 and B a value of 3 will generate white. On a 480Z system, combinations of red, blue and green will generate the three colours yellow, magenta or cyan. The third form with no arguments restores the default values set up by RESOLUTION.

Example:

```
10 CALL "RESOLUTION",1,4
20 DATA ...specifying 16 colours
30 FOR I=0 TO 15
40 READ N
50 CALL "SETCOL",I,N
60 NEXT I
70 CALL "VIEW",0
```

sets mode MR4, then loads the colour lookup table to display page 0, the view being determined by the DATA statement.

UPDATE

Forms:

```
CALL "UPDATE",P,V    (all modes)
CALL "UPDATE",V      (HR 1)
CALL "UPDATE",P      (MR 4)
```

Purpose: To make page P, view V accessible for modification by PLOT, LINE or FILL.

Remarks: In mode HR2 and EHR there is only one page and view and this call is not used. In all MR modes there are two separate pages, numbered 0 and 1. In modes HR1 and MR2 there are two views, numbered 0 and 1, and in mode MR1 there are four views, numbered 0 to 3.

After a call to RESOLUTION, page 0 view 0 is selected for update; that is, accessible to PLOT, LINE and FILL. Calling UPDATE changes the page and/or view that is accessible. Similarly, calling DISPLAY selects the page and view that is displayed.

P may be omitted in mode HR1 and V in mode MR4.

Example:

```
10 CALL "RESOLUTION,1,1
20 CALL "UPDATE",1,3
```

set mode MR1, then select page 1, view 3 for update.

VIEW

Forms:
```
CALL "VIEW",P    (all modes)
CALL "VIEW"      (HR only)
```

Purpose: To load the colour lookup table with values set up in memory by SETCOL, allowing the selection of non-standard views (see also SETCOL).

Remarks: The VIEW call copies the colour lookup table from memory to the actual table on the HRG board and selects an MR page. The table in memory is initialized by RESOLUTION and modified by COLOUR and SETCOL. Use of SETCOL followed by VIEW allows the selection of non-standard views; it also allows the loading of the hardware table to be deferred until all colours have been set up. This CALL has no effect in EHR.

The argument P is the medium resolution page required and should be 0 or 1. It may be omitted in modes HR1 and HR2.

CAUTION: it is important to set up all four (HR) or sixteen (MR) intensity values by calling SETCOL before calling VIEW, otherwise undefined values will be loaded into the colour lookup table. This applies even in modes HR1 and MR1, and is in contrast to calling COLOUR in these modes, where only intensities 0 and 1 can be defined.

Example:
```
10 CALL "RESOLUTION",0,1
20 DATA 0,200,200,0
30 FOR I=0 TO 3
40 READ N
50 CALL "SETCOL",I,N
60 NEXT I
70 CALL "VIEW"
```

sets up HR1, then loads the colour lookup table to display the differences between view 0 and view 1.

CHAPTER 17

HIGH RESOLUTION GRAPHICS LEVEL 2

The facilities described in this chapter are provided in Level 2 support of High Resolution Graphics (HRG) from Extended BASIC Versions 5 and 6. They are in addition to the Level 1 routines described in chapters 15 and 16.

The additional features permit:

- Dumping rectangular areas of the HRG screen to an Anadex DP 9501 printer or an Epson MX type III/RX/FX series printer.
- The enlargement and shading patterns may be one of a pre-defined set, or they may be user-defined.
- Plotting character-strings on the HRG screen in any of four orientations.
- Copying one rectangular area of the HRG screen to another.
- Reading the logical intensity of a point on the HRG screen.

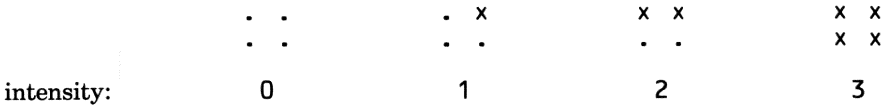
The first section of this chapter is a brief introduction to the new facilities of HRG Level 2. Then there is the main section of formal definitions which is layed out in the same way as in the previous chapter. Finally there are some example programs.

INTRODUCTION

Dumping to a printer

NOTE: "x" denotes a spot and "." denotes a blank in this discussion.

The colour/intensity of each point present on the screen is printed on the paper as a pattern of dots. This pattern of dots, known as a shading pattern, is normally chosen to give a similar 'brightness' to the pixel it is representing. For example, if the graphics screen is set to HR2 mode ("RESOLUTION",0,2), then four colours/intensities are possible. These are set up by default ranging from intensity 0 (dimmest) to intensity 3 (brightest). So for this mode, we require 4 different shading patterns and we might use the following:



In this case, each time a pixel from the screen is printed, it is represented by one of the four shading patterns shown earlier.

The CALL "PATSIZE" command is used to define what pattern size will be used for printing. Shading patterns can be defined using a call to the SHADING routine.

HRG characters

The "normal" ASCII characters displayed on the computer screen are generated by circuitry contained on one of the boards in the computer (the video controller board). The characters making up the text are normally called 'hardware characters'. The high resolution graphics board can also produce textual information using the STPLOT function. In this case the characters are defined by software, so they are called 'software characters' or 'HRG characters'. An HRG character is displayed by filling pixels on the HRG screen to make up the shape of the character.

When you switch on, this software character set is loaded into RAM; the characters are very similar to the 32 special characters and 96 ASCII characters of the hardware characters (see Appendix B of the 480Z Users Guide). The software characters may be accessed with the CALL "STPLOT" command and the shape of any of them may be re-defined by the CALL "DEFCHR" command.

NOTE: No call can be executed in HRG Level 2 unless the RESOLUTION routine has been called at least once since you loaded BASIC.

CHARSIZE

Form: `CALL "CHARSIZE", XE,YE`

Purpose: To define the magnification of the characters to be displayed by a call to the STPLOT routine.

Remarks: The XE argument of a call to the CHARSIZE routine defines the magnification in the X-direction. The YE argument defines the Y-direction magnification. Both XE and YE must be in the range 1 to 255.

A value of zero will produce the error message:

```
Illegal arg
```

The value of XE and YE are reset to 1 on any call to the RESOLUTION routine.

Example: Try adding the line:

```
25 CALL "CHARSIZE", 3, 3
```

to the example program given in the DUMP definition.

COPY

Form: `CALL "COPY", X1, Y1, X2, Y2, XD, YD`

Purpose: To move a copy of a rectangle from one area of the screen to another.

Remarks: A call to the COPY routine moves a copy of the rectangle defined by lower left corner (X1, Y1) and upper right corner (X2,Y2). the destination of the copy is defined by the lower left corner (XD,YD).

If any of the coordinates X1, Y1, X2 or Y2 are outside the area of the screen for the current resolution, then they will be adjusted so that any value outside is placed in range.

Note that the COPY routine works with the view/page that is currently being updated, not necessarily the view/page that is being displayed.

Example: In HR mode, the following statement:

```
CALL "COPY", 220,20,120,80,120,100
```

moves a copy of the rectangle upwards and to the right.

DEFCHAR

Form: `CALL "DEFCHAR",C,B1,B2,B3,B4,B5,B6,B7,B8`

Purpose: To define the characters to be displayed by a call to the STPLOT routine.

Remarks: The C argument of the call to the DEFCHAR routine is the ASCII value (in the range 0 to 127) of the character to be defined.

The arguments B1 to B8 are the bytes that make up the character. B1 refers to the top row of the character, B2 to the second, B3 to the third, and so on.

The decimal values of B1, B2, ..., B8 are converted to binary numbers. For example, the decimal 8 is converted to the binary 00001000. Any bit set at 1 is converted to a displayed dot. Any bit set at zero is converted to a blank. The least significant binary digit is on the right.

Example: `CALL "DEFCHAR, ASC(".")",3,4,8,8,8,8,16,96`

Redefines the full stop character to an integral sign.

	Decimal	Binary	Character
B1	3	00000011	XX
B2	4	00000100	.....X..
B3	8	00001000	....X...
B4	8	00001000	....X...
B5	8	00001000	....X...
B6	8	00010000	...X....
B7	16	00010000	...X....
B8	96	01100000	.XX.....

Two further examples of character definition are:

CAPITAL 'A'

	Decimal	Binary	Character
B1	24	00011000	...XX...
B2	36	00100100	..X..X..
B3	66	01000010	.X....X.
B4	66	01000010	.X....X.
B5	126	01111110	.XXXXXX.
B6	66	01000010	.X....X.
B7	66	01000010	.X....X.
B8	66	01000010	.X....X.

DOLLAR SIGN

	Decimal	Binary	Character
B1	0	00000000	....
B2	8	00001000	....X...
B3	63	00111111	..XXXXXX
B4	72	01001000	.X..X...
B5	62	00111110	..XXXXX.
B6	9	00001001	....X..X
B7	126	01111110	.XXXXXX.
B8	8	00001000	....X...

DUMP

Form: `CALL "DUMP", X1, Y1, X2, Y2, [,D]`

Purpose: To copy a rectangle of the screen to the printer.

Remarks: The rectangle to be copied is specified by the bottom left corner (X1, Y1) and the top right corner (X2, Y2).

The fifth parameter, D, is optional and determines the method of printing. The rectangle can be printed either across the paper or along the paper, depending on the value of D. Its value also determines whether the background colour on the screen will be printed as black or white. White is the default colour. If black is chosen then all the shading patterns will be reversed:

D=0 — Causes the x-axis to be in the direction of paper travel with the background colour printe as white. (the reverse of what is shown on the screen) This is the default method.

D=1 — Causes the x-axis to be across the width of the paper with the background colour printed as white (the reverse of what is shown on the screen).

D=2 — Causes the x-axis to be in the direction of the paper travel with the background colour printed as black.

D=3 — Causes the x-axis to be across the width of the paper with the background colour printed as black.

Note that the DUMP routine works with the view/page that is currently being updated, not necessarily the view/page that is being displayed.

Before you attempt to make a copy, ensure that:

1) there is a suitable printer attached to the 380Z/480Z. Suitable printers are Anadex 9501 and Epson MX type III/RX/FX series printers

2) you have set up the printer option with the BASIC PRINTER command (and, if on a network, have selected either NETWORK PRINTER or LOCAL PRINTER)

3) you have selected the correct printer with the CALL "PRINTER" command (see below)

Indications that the printer has not been set up correctly are:

- A large number of '@' signs produced on the screen.
- Output of control codes to the screen and the beeper sound.
- A large number of '@' signs produced on the printer.

If you are on a network you may find it advisable to select LOCAL PRINTER before calling DUMP. This will avoid generating very large spooler files, which typically will not fit on the disc. If you must use a network printer, we recommend that you do not use a pattern size greater than 1×1.

Note also that if dumping over a network the output may be incomplete or incorrect, with blank lines occasionally inserted. This is due to the way the network spooler interprets some of the control codes which are to be sent to the printer. This applies particularly to users of Epson printers.

Example: The following program is a simple example of dumping a picture (the CALL "PRINTER" command is defined later in this chapter):

```
  5 PUT 12     :REM Clear the screen of text
 10 PRINTER 3 :REM Select parallel printer
 20              REM - change above line to
 25              REM suit your printer
 30 CALL "RESOLUTION",0,2
 40 CALL "PRINTER",1 :REM Select EPSON
 45              REM printer, or change to
 50              REM CALL "PRINTER",0 for
 55              REM an ANADEX printer
 60 ITEN=0
 70 REM ---------------
 80 FOR Y=0 TO 120 STEP 30
 90 CALL "FILL",0,Y,30,Y+28,ITEN
100 ITEN = ITEN+1
110 NEXT Y
120 REM ---------------
130 CALL "DUMP",0,0,30,120,0
140 CALL "CLEAR"
```

PATSIZE

Form: `CALL "PATSIZE", P, Q`

Purpose: To define a pattern size to be used for each point from the screen when dumping to the printer.

Remark: The size of each shading pattern, and hence the size of the printed picture can be changed with a call to the PATSIZE routine. The argumets, P and Q, give the size of the shading pattern; P is the x component and Q is the y component. The maximum value for both P and Q is 6.

If no call to the PATSIZE routine is made, then a default size and pattern (related to the current resolution) is used:

Resolution	Maximum Number of logical Intensities	Default size of shading pattern
0,2	4	3x3
0,1	2	3x3
1,4	16	4x4
1,2	4	3x3
1,1	2	3x3
2,1	2	3x3

and any call to the RESOLUTION routine will produce a size and pattern according to this table. The default pre-defined shading patterns are shown below.

There are other pre-defined shading patterns which are automatically selected when the following values of P and Q are chosen:

Maximum number of Logical intensities	Sizes of shading pattern which have pre-defined patterns	
2	1x1	
	2x2	
	3x3	(default)
	4x4	
4	2x2	
	3x3	(default)
	4x4	
16	4x4	(default)
	6x6	

Any other values of P and Q will give shading patterns of blanks and the previous shading patterns will be lost. New shading patterns can be defined using a call to the SHADING routine.

Pre-defined shading patterns

With 2 logical intensities:

Colour 1 will be no dots. Colour 0 will be all dots printed in the matrix

With 4 logical intensities:

Pattern size 2×2

Intensity	0	1	2	3
	. . . .	. . . x	. x . x	x x x x

Pattern size 3×3

Intensity	0	1	2	3
	. . . .	. x . x . x . x .	x x x x . x x x x	x x x x x x x x x

Pattern size 4×4

Intensity	0	1	2	3
	. . . .	. x x . x . . x x . . x . x x .	x x x x x . . x x . . x x x x x	x x x x x x x x x x x x x x x x

With 16 logical intensities

Pattern size 4x4

Intensity

0	1	2	3
....	...x	....	x..x
....	....	x..x	....
....	....	..x.	....
....	x...	....	x..x

4	5	6	7
.xx.	x..x	xx..	...x
x...	.x..	...x	x..x
x..x	..x.	x...	x..x
....	x..x	x.xx	.xx.

8	9	10	11
x.xx	xx.x	xx.x	xxxx
...x	x..x	x.xx	x..x
.x..	x..x	.xx.	x..x
xxxx	x.xx	xx.x	xxxx

12	13	14	15
xxxx	xxxx	xxxx	xxxx
x..x	xx.x	x.xx	xxxx
xx.x	x.xx	xxxx	xxxx
xxxx	xxxx	xxxx	xxxx

Pattern size 6x6

Intensity 0

```
......
......
......
......
......
......
```

1

```
......
.X..X.
......
......
.X..X.
......
```

2

```
X....X
......
...X..
..X...
......
X....X
```

3

```
.X..X.
X....X
......
......
X....X
.X..X.
```

4

```
.X..X.
XX..XX
......
......
XX..XX
.X..X.
```

5

```
.XX.X.
X....X
X....X
X....X
X....X
.XXXX.
```

6

```
.XXXX.
X....X
.X..X.
.X..X.
.XXXXX.
X....X
```

7

```
X.X.X.
.X.X.X
.X.X.X
X.X.X.
.X.X.X
X.X.X.
```

8

```
XXXXXX
X....X
X....X
X....X
X....X
XXXXXX
```

9

```
XXXXXX
X....X
X.XX.X
X.XX.X
X....X
XXXXXX
```

10

```
XX..XX
X.XX.X
.XXXXX
XXXXX.
.X..X.
XX..XX
```

11

```
XXXXXX
XX..XX
X.XX.X
X.XX.X
XX..XX
XXXXXX
```

12

```
XXXXXX
X.XX.X
XX.XXX
XXX.XX
X.XX.X
XXXXXX
```

13

```
XXXXXX
XXXXXX
XX..XX
XX..XX
XXXXXX
XXXXXX
```

14

```
XXXXXX
X.XXXX
XXXXXX
XXXXXX
XXXX.X
XXXXXX
```

15

```
XXXXXX
XXXXXX
XXXXXX
XXXXXX
XXXXXX
XXXXXX
```

Note that a small pattern size must be used when using an 80 column width printer as a full screen dump will not fit on the paper with larger sizes.

Example: The following statement:

```
CALL "PATSIZE",3,3
```

will define the pre-defined 3x3 patterns (with 4 logical intensities) as the ones to be used when dumping to the printer.

PRINTER

Form: `CALL "PRINTER", T[, W]`

Purpose: To define which type of printer will be used for output.

Remarks: The CALL "PRINTER" command enables output to either an Anadex DP9501 printer or an Epson MX-type III/RX/FX printer. The first argument, T, defines the printer type. If T is 0, an Anadex is to be used; if T is 1, an Epson is to be used. The default is T=0 (Anadex).

The optional second argument, W, defines the print mode. When using an Anadex, this argument has no effect, but with an Epson printer the following applies:

W = 0 Normal density bit image print

W = 1 Dual-density bit image print, half the width of normal density

The default is W = 0 (normal density)

Example: See the example program in the DUMP definition.

RDOUT

Form: `CALL "RDOUT", X, Y, VARADR(V)`

Purpose: To read the logical intensity at a point on the screen and to return this intensity in the variable V.

Remarks: The CALL "RDOUT" function is similar to the BASIC low resolution graphics function, POINTS. The logical intensity at the point given by X and Y will be returned in the variable, V.

Note that the RDOUT routine works with the view/page that is currently being updated, not necessarily the view/page that is being displayed.

Example:
```
CALL "RDOUT", 30, 30, VARADR(V)
```

returns the logical intensity of the specified point on the screen in the variable, V.

SHADING

Form: `CALL "SHADING", I, VARADR(A$)`

Purpose: To define a shading pattern for a logical intensity, I.

Remarks: The second argument contains the shading information in A$, which is divided into groups of characters (0 and 1). For example, in a 4x4 shading pattern, A$ is divided into four groups of four characters.

Each group corresponds to one row of the shading pattern. The left group is the top row and the right group is the bottom row. The left character in each group is the left position of that row.

An error message is produced if the string length is incorrect for the current shading patern size (Invalid Pattern Length) if any element of the string is not 0 or 1 (Illegal Shading Pattern), or if I is not a possible logical intensity for the current resolution (Illegal argument).

A shading pattern is lost whenever a call to a RESOLUTION or PATSIZE function is made.

Example: As an example,

```
A$="0001001001001000"
```

gives the following shading pattern:

```
...x   "x" denotes a spot
..x.
.x..   "." denotes a blank
x...
```

So 0 represents a blank and 1 represents a spot.

STPLOT

Form: `CALL "STPLOT", X, Y, VARADR(A$), I[, D]`

Purpose: To display a string of characters on the screen

Remarks: The first two arguments, X and Y, define the starting point at which the string, A$, will be displayed. The argument, I , is the logical intensity, and the optional argument, D, is the direction in which the string will be plotted:

D=0 normal horizontal characters (this is the default value)
D=1 sidways up the screen
D=2 upside down
D=3 sideways down the screen.

If a character in the string is outside the range 0-127, it will be folded back into that range. For example, a character with ASCII value 255 would be mapped onto 127. A direction outside the range 0-3 is folded back.

Example:

```
 5 REM Display text with HRG character set
10 PUT 12
20 CALL "RESOLUTION", 0, 2
30 A$="HELLO WORLD"
40 CALL "STPLOT", 30, 30, VARADR(A$),3
```

displays:

```
HELLO WORLD
```

on the screen

EXAMPLE PROGRAMS

1) EXCOPY

This simple program is designed to show the speed of the COPY function.

```
 10 REM EXCOPY
 20 PUT 12
 30 CALL "RESOLUTION", 0, 2
 40 CALL "FILL", 0, 0, 40, 40, 3
 50 CALL "FILL", 8, 8, 32, 32, 0
 60 CALL "FILL", 17, 17, 36, 26, 2
 70 FOR J=30 TO 150 STEP 30
 80 CALL "COPY", J-30, J-30, J+10, J+10, J, J
 90 NEXT J
100 END
```

2) SINE

This program plots and labels a sine graph, showing some of the features of the STPLOT function. It then prints the graph on the printer, with a white background. The x-axis runs across the width of the paper, and a pattern size of 2×2 with default shadings is used. If you are using a network then we suggest that you use a local printer.

```
  3 LOCAL PRINTER 3 :REM  Change this line to
  5                   REM suit your printer
 10               REM  SINE shows a use of STPLOT,
 15               REM  and demonstrates DUMP.
 20 PUT 12
 30 LET X$="X" : LET Y$="y"
 40 LET A$=" " : LET P$=CHR$(21)
 50    REM P$ is the HRG character pi.
 60 LET P2$="2"+P$ : LET P3$="3"+P$
 70 LET I$="1" : LET I1$="-1"
 80 CALL "RESOLUTION",0,1
 85 CALL "PRINTER",1 :REM  Change this line to
 86                   REM  CALL "PRINTER",0 for
 87                   REM  an ANADEX printer.
 90 CALL "OFFSET", -8, -96
100    REM plot sine wave.
110 LET A=16*ATN(1)/318
120 FOR X=0 TO 318
130 LET Y=50*SIN(A*X)
140 CALL "PLOT", X, Y, 1
150 NEXT X
```

```
160     REM plot and label axes.
170 CALL "PLOT", 0, -96, 0
180 CALL "LINE", 0, 96, 1
190 CALL "STPLOT", -4, 88, VARADR(A$), 1
200 CALL "STPLOT", 8, 88, VARADR(Y$), 1
210 CALL "PLOT", 0, 0, 0
220 CALL "LINE", 311, 0, 1
230     REM line 240 plots "?" on its side.
240 CALL "STPLOT", 303, 4, VARADR(A$), 1, 3
250 CALL "STPLOT", 302, 8, VARADR(X$), 1
260     REM  Draw a heading and mark points
265     REM  on each axis.
270 LET B$="Graph of y=sin x"
280 CALL "STPLOT", 3, 70, VARADR(B$), 1
290 CALL "PLOT", -2, 49, 0
300 CALL "LINE", 2, 49, 1
310 CALL "STPLOT, 5, 45, VARADR(I$), 1
320 CALL "PLOT", -2, -50, 0
330 CALL "LINE", 2, -50, 1
340 CALL "STPLOT", 5, -53, VARADR(I1$), 1
350 CALL "STPLOT", 79, 4, VARADR(P$), 1
360 CALL "STPLOT", 2*79-16, 4, VARADR(P2$), 1
370 CALL "STPLOT", 3*79, 4, VARADR(P3$), 1
380     REM Set the shading pattern size to be 2x2
390 CALL "PATSIZE", 2, 2
400     REM Dump the screen to the printer.
405     REM Assumes correct type has been set up.
410 INPUT "Press <RETURN> when ready to dump",A$
420 CALL "DUMP", -8, -96, 310, 95, 1
430 END
```

CHAPTER 18

MACHINE AND ASSEMBLY LANGUAGE SUPPORT

This chapter is mainly of interest to those who wish to add machine-language routines to BASIC for special requirements. Reasonable familiarity with Z80 Assembly Language is assumed.

Extended BASIC has some features which allow direct access to main memory during the execution of BASIC programs. Control can be passed to machine-language routines to perform special functions. An example of this is the use of the CALL command to add support to BASIC for high resolution graphics (see chapter 15).

Small routines can be put into memory from within BASIC programs using the POKE command. Larger routines, developed with a Z80 Assembler, can be loaded from disc files. When running BASIC programs, the USR and CALL commands allow the BASIC interpreter to pass control to these machine-language routines to perform special requirements.

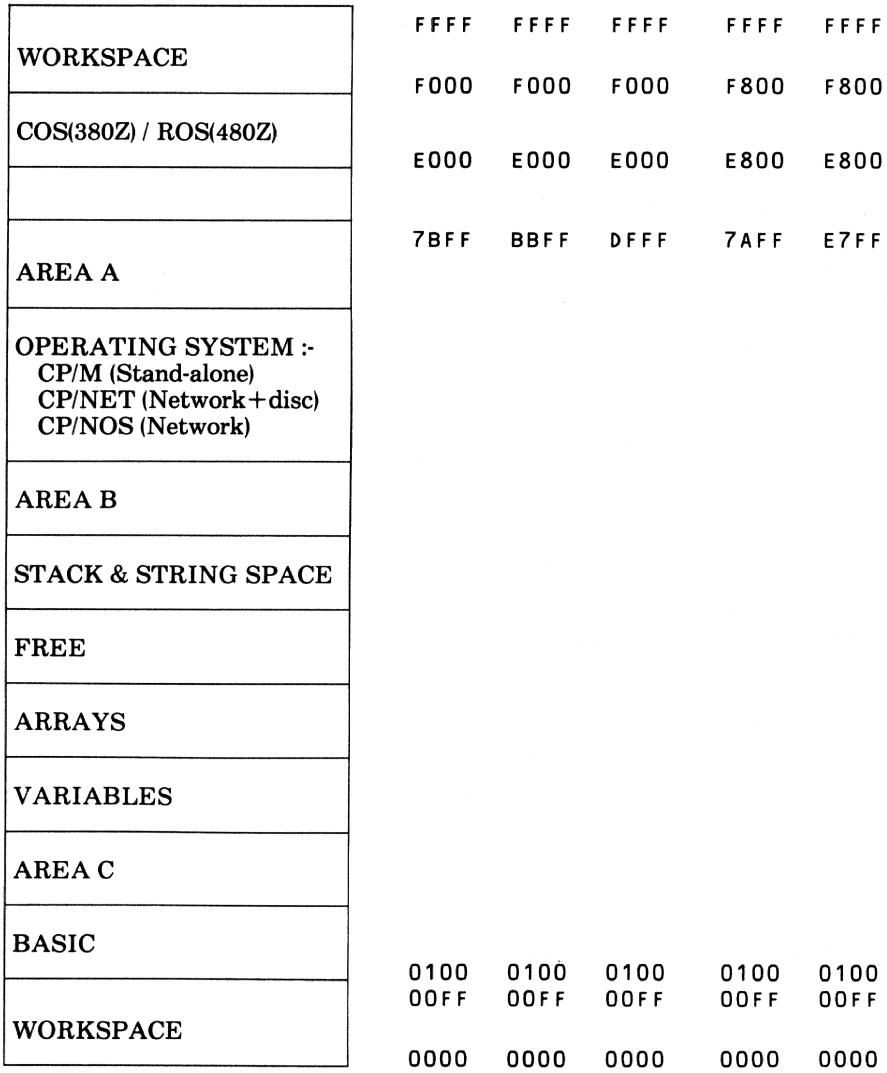
The first part of this chapter gives details of the 380Z and 480Z memory layout. Guidance is given about the areas of memory that should be used for storing machine-language routines that will be called from BASIC programs. The next section defines the available BASIC commands, which are:

PEEK — return the contents of a memory location
POKE — store a value in a memory location
INP — return a value from an input port
OUT — send a value to an output port
WAIT — wait for an input status bit
VARADR — return the address of a variable
USR — call a user-supplied function
CALL — call a user-supplied routine.

The last part of the chapter gives instructions on machine-language initialization and on adding and saving machine-language routines; some examples of machine-language routines and useful BASIC programs are provided.

When a requirement does arise that cannot be met by the BASIC language, you are urged to use the mechanisms described in this chapter. Do not attempt to modify the BASIC interpreter itself; this will lead to programs that will be difficult to share with other users. Programs from modified interpreters cannot be maintained if the interpreter is changed. You are reminded that the Research Machines software licence agreement restricts the use of the licenced software to a single computer. You break this agreement if you give a colleague a program plus a modified interpreter.

MEMORY LAYOUT



All addresses are hexadecimal.

Addresses in the above diagram that appear too high for the specified memory size are mapped to the appropriate locations.

WHERE TO PUT MACHINE-LANGUAGE ROUTINES

A machine-language routine that is to be called from BASIC may reside in one of three areas, designated A, B, and C in the memory map diagram.

Area A is a block of memory immediately above the operating system, and is normally of zero size. On a stand-alone system the MOVCPM and SYSGEN programs can be used, if required, to move the operating system down (by 1K increments) and so to reserve space for user-supplied subroutines. For example, the command:

```
A>MOVCPM 30 *
```

in a 32K 380Z, followed by SYSGEN, would generate a 30K operating system and make available memory at addresses 7800 to 7BFF hex. User-supplied routines can be installed in this area by POKE commands, or by using the DDT program before starting BASIC. The MOVCPM, SYSGEN and DDT programs are described in the relevant User Guide.

Area B is at the top of the area available to BASIC, immediately below the operating system, and is termed the cache memory. Cache memory is reserved from within a BASIC program by means of the CLEAR command (see chapter 6). Machine language routines can be placed in this area only by the use of POKE commands. A way of finding the address limits of cache memory is given below.

Area C is between the BASIC interpreter and the BASIC program area. Any user-supplied machine-language routines (debugged) that are required to be loaded with BASIC should be installed in this area.

In general, small routines that are put in memory from BASIC with the POKE command should be stored in cache (Area B). Larger routines that are developed with a Z80 Assembler should be tested in Area A. If you want them to become a permanent part of BASIC, you must re-assemble them for Area C then load them with BASIC (details of linking machine-language routines are given later in this chapter).

Finding the address limits of cache memory

The start of cache is found from ENDMEM, which contains the address of the last byte of RAM used by BASIC, and HIMEM, which contains the address of the first byte of the operating system. Thus the current size of cache memory is given by (HIMEM)—(ENDMEM)—1 where parentheses mean "contents of".

ENDMEM is normally found in locations 011C and 011D hex, but check the actual address as given in the Release Note.

HIMEM is found in locations 0006 and 0007 hex.

The following program displays the current start address and size of cache memory, the size of string space, and the amount of free memory remaining (the PEEK command is defined below):

```
10 LET EM=PEEK(&11C)+256*PEEK(&11D)
20 LET HM=PEEK(6)+256*PEEK(7)
30 PRINT "CACHE START ="; EM+1
40 PRINT "CACHE SIZE ="; HM-EM-1
50 PRINT "STRING SPACE="; FRE(X$)
60 PRINT "FREE MEMORY ="; FRE(X)
```

You may find it interesting to run this program a number of times, issuing different forms of the CLEAR command between each run.

FUNCTION AND COMMAND DEFINITIONS

PEEK `10 LET A=PEEK(B)`

This function returns the contents of the memory location at the address specified by the argument. The argument must lie in the range 0 to 65535 (0 to FFFF hex). The function returns the contents as an 8-bit binary number which lies in the decimal range of 0 to 255.

POKE `10 POKE A, 0`

The POKE command stores its second argument in memory at the address specified by its first argument. The stored second argument must be in the range 0 to 255 (0 to FF hex); the first address argument must be in the range 0 to 65535 (0 to FFFF hex). The example in the heading stores 00 hex (the no operation instruction, NOP) at address A. Another example of the use of the POKE command is given in the definition of the CALL command.

The POKE command is best used in conjunction with the cache memory, where small machine-language routines can be stored. cache memory can also be used to preserve integer values between programs.

INP `LET V = INP(P)`

This function returns the contents of the input/output (I/O) port specified by the argument. The port address must lie in the range 0 to 65535; an 8-bit binary value is returned (0-255). The example reads value V from port P. Memory-mapped ports cannot be accessed.

Useful port addresses are provided in the relevant Information File (380Z or 480Z).

OUT

```
OUT P, V
```

The OUT command writes the 8-bit binary value (in the decimal range of 0 to 255) of its second argument to the I/O port specified by the first argument (which must be in the range 0 to 65535). The example writes value V to port P. Memory-mapped ports cannot be written to using the OUT command.

WAIT

```
WAIT P, M, X
```

The WAIT command causes BASIC to pause until a specified bit pattern is read from an I/O port. The first argument specifies which I/O port must be read; the second argument specifies which bit or bits must be tested; the third argument specifies that BASIC must wait as long as the specified bit(s) are set to this argument. The third argument can be omitted, in which case zero is assumed.

The first argument must lie in the range 0 to 65535. The second and third arguments must be in the range 0 to 255. The example tests port P and makes BASIC wait while the bit(s) specified by M remain in state X.

You must think of M and X as eight-bit binary numbers. For example, if M is decimal 2, it is written in binary notation as 00000010. This specifies that bit 1 must be tested (bit 0 is on the right and bit 7 is on the left). The statement:

```
10  WAIT &E9, 2
```

makes BASIC wait as long as binary bit 1 of port E9 is at zero. When bit 1 changes to 1, BASIC continues. The third argument was omitted, so zero was assumed. Conversely, the statement:

```
10  WAIT &E9, 4, 4
```

waits as long as binary bit 2 of port E9 is set at 1. When it becomes zero, BASIC continues.

(The mechanism involves Boolean operations: read the port; exclusive OR its contents with X; AND the result with M; repeat until the result is non-zero)

If necessary, more than one bit can be tested simultaneously.

For example:

```
10  WAIT &C9, 3, 2
```

makes BASIC wait while bit 0 is zero and bit 1 is one. When either bit 0 or bit 1 changes, BASIC continues.

VARADR `LET A=VARADR(V)`

The VARADR function returns the address of a variable (numeric or string), or an element of a numeric or string array. The example assigns the address of variable B to A.

Binary Floating Point Notation

In Research Machines Extended BASIC, numeric variables are held in four bytes using the binary floating point notation. The decimal number 0.5 is represented by the four-byte number:

```
00  00  00      80
   fraction     exponent
```

The three left-hand bytes represent the fraction and the right-hand byte represents the exponent.

The exponent is a power of 2; for ease of calculation, 128 (80 hex) is added to the exponent so that it is always positive. The fraction lies in the range 0.5 to 0.99...., so its most significant bit is always set. Therefore, this bit is not stored; instead the bit position is used to store the sign of the number, zero for a positive number and one for a negative number. The least significant byte of the fraction is on the left of the three bytes. The most significant byte is on the right (next to 80 in the above example which can be translated to + 0.5E0 or 0.5).

Some further examples are:

```
0 0   0 0   8 0   8 0 = − 0.5
0 0   0 0   4 0   8 0 = + 0.75
C F   0 F   4 9   8 2 = + 3.14159
```

If the exponent is zero, the number represents zero, whatever value the fraction has.

For a numeric variable, each of the four bytes occupies a memory address. The VARADR function returns the address of the left-hand byte. The remaining bytes occupy the next 3 higher memory addresses, with the exponent byte in the highest address.

For a string variable, the address returned by the VARADR function is that of a four-byte "description block" which gives information about the string. The first byte contains the string length (0 to 255 characters), the second byte is always zero and the third and fourth bytes contain the address of the string (with the less significant byte first).

The VARADR function can be used in a CALL or USR instruction to pass the address of a variable to a machine-language routine.

The latter can then return information to the BASIC program by modifying the value stored in the variable.

CAUTION: The address of an array element will change whenever a simple variable is assigned to it for the first time. This is because BASIC stores simple variables below array variables in an adjoining list.

Also, the actual address of the string in a string variable description block will change from time to time as BASIC re-organizes its string space. If a literal string has been assigned to the string variable, the address in the description block may point into the program text.

For both these reasons you must never increase the length of a string in a machine-language routine, though you can decrease its length.

USR

```
LET V = USR(B)
```

This function allows BASIC to obtain a value from an external user-provided, machine-language routine. The example returns a value from the machine-language routine. This value is assigned to V; an argument is passed to the routine in the variable, B.

The transfer address must be set up before the USR function is called otherwise the error message "Illegal function" is displayed. This address must be set up at address USR (the address of this vector is given in the Release Note) which is the address of a jump instruction to your machine-language routine.

The routine address must be placed in (vector +1) and (vector +2) with the less significant byte first. The routine does not have to use the argument passed to it.

Within your routine, the value of the argument B can be obtained and converted to 16 bits by a call to address USRVF. On returning from this call, the value is in register pair DE. The user can return a sixteen-bit number to BASIC as the value of the USR function by loading it into register pair AB (register A containing the more significant byte) and calling address USRVT. Both these calls are optional; if no call to USRVT is made then the value returned is undefined. Finally control is returned to BASIC by a RET instruction (C9 hex). The code segment may modify any Z80 register. The addresses of USRVF and USRVT are given in the Release Note.

The USR function is a relatively simple method of passing values between BASIC and machine-language routines. A useful example of the use of the USR function is given in the Examples section.

CALL

```
CALL A,V1,V2    CALL "ST16",V1,V2
```

The CALL command is used to call a machine-language routine.

The first numeric or string argument gives the address of the routine. This argument can be followed by any number of numeric arguments (separated by commas) which are converted to sixteen-bit integers available to the routine. In the above examples, two arguments, the current values of variables V1 and V2, are passed.

If the first argument is a number, numeric variable, or numeric expression, the argument represents the address of the routine to be called (as in the first heading example). The only exception to this rule is that a call to address 0 is ignored because it is likely that the address variable has been cleared (after, for example, an EDIT command).

If the first argument is a string constant, variable, or expression, then BASIC searches through a list of names of routines, supplied by you, for a match. This list of nodes is formatted as in the following example:

```
DEFW    NEXTNODE    ;Pointer to the next node
DEFB    NAMELEN     ;Length of the name
DEFM    'NAME'      ;Name of the routine
DEFW    ROUTINE     ;Address of the routine
           ↑
        your names
```

If a match is found, the routine is called; if no match is found, the error message "Name not found" is displayed and the program stops.

The address of the first node must be placed in the symbolic location SUBPTR (the address of this location is given in the Release Note). In the last node, the pointer to the next node must contain the previous contents of SUBPTR.

This node organization lets you add further routines without a knowledge of the other nodes. The chain of nodes must be followed from SUBPTR until the zero pointer is encountered.

On entry to the routine, the values of the arguments following the first argument are on the Z80 stack. They may be popped (taken) off in reverse order. In the heading examples, V2 is popped off first. Register C contains the count of arguments in the stack; this count can be used by the routine to determine how many arguments to pop.

In addition, register pair HL contains the address on the stack of the return address to BASIC, so the routine may either pop the correct number of arguments off the stack, or load the stack pointer from register HL (LD SP,HL), then perform a return from routine

instruction (RET) to return control to BASIC. Within the routine, any register may be modified.

Returning values to BASIC with the CALL command is best accomplished by using the entry point SAVVAR (its absolute address is given in the Release Note). When called, SAVVAR converts the sixteen-bit quantity in register pair AB into floating point, and stores it in the variable whose address is in register pair HL. For example, the following routine:

```
10 LET A=0
20 CALL ADDR,A,VARADR(A)

ADDR:   POP   HL      ;Address of A
        POP   DE      ;Value of A
        INC   DE      ;Add one to A
        LD    A,D
        LD    B,E
        CALL  SAVVAR  ;Store it
        RET           ;Back to BASIC
```

assigns 1 to A. Note that SAVVAR destroys the contents of all the Z80 registers.

Examples of routines accessed by the CALL command are the High Resolution Graphics facilities, described in Chapters 15-17.

MACHINE-LANGUAGE ROUTINE INITIALIZATION

When BASIC is started up (by responding with BASIC to the A> prompt, for example), control is passed to the BASIC initialization procedure. During this procedure BASIC calls a further initialization procedure (INITSUB) which normally returns immediately to the BASIC initialization procedure. You may modify the INITSUB procedure to suit your own requirements.

The INITSUB procedure occupies three memory locations. The first contains the Z80 instruction, RET (C9 hex), and the second and third locations contain NOP instructions (00 hex). These instructions cause an immediate return to the BASIC initialization procedure.

As previously mentioned, you may change these instructions. The usual modification is to set the three memory locations to contain a jump instruction followed by the two-byte address of the required machine-language routine. Such a routine might perform further initialization for an additional peripheral and indicate the presence of that peripheral. The version number message produced by the high resolution graphics (HRG) routines, for example, is produced in this way following an initialization test for the presence of the HRG board.

There are two distinct methods for returning to the BASIC initialization procedure from the machine-language initialization procedure:

1. If the routines are attached to a version of BASIC with no other machine-language routines, then terminate your initialization procedure with a RET instruction.

2. If there are already routines attached to BASIC which have modified the contents of the 3 INITSUB locations, then the machine-language initialization procedure should end by executing the previous INITSUB code.

Below is an example of the instructions required to patch an initialization routine onto a BASIC program without any machine-language routines attached:

```
ORG INITSUB
JP  BEGIN  ;on initialization jump to label BEGIN
```

The above two instructions modify the contents of the three INITSUB locations. An example of a machine-language initialization procedure is given below. It uses the subroutine termination method (see 1. above):

```
        ; Initialization Code starts here

BEGIN:  LD      HL,INITMSG          ;For example
        EMT     MSG
        .
        .       ;Any other initialization code

        RET     ;Return to BASIC.
                ;If the routine is being attached to
                ;a version of BASIC which already
                ;has machine-language routines
                ;attached, then the RET instruction
                ;should be replaced by the
                ;code originally at INITSUB
```

ADDING AND SAVING MACHINE-LANGUAGE ROUTINES

There is a utility program on the Extended BASIC distribution disc called CODE.BAS to provide easy linking of machine-code routines with the BASIC interpreter. The program produces a ZASM source listing with necessary ORG and EQU statements which will automatically "include" your source file(s) when running ZASM. If you have more than one routine to link into BASIC then arrange them in one or more files and enter their names into the CODE.BAS program as separate segments.

A run of the procedure to link a routine is shown below.

Example:

This example shows how to attach to BASIC the two routines TEST1 and TEST2 which will be accessed from the interpreter by the commands CALL "TEST1" and CALL "TEST2". The source code for these routines is held in the files BASTEST1.ZSM and BASTEST2.ZSM shown here:

```
*H BASTEST1

        OUTC   =          1
        FFEED  =          12

;Routine to be attached to BASIC which clears
;screen leaving cursor at bottom left.

START1:LD      A,FFEED
        EMT    OUTC

        RET

*H BASTEST2

        OUTC    =          1
        US      =          31

;Routine to be attached to BASIC which clears
;screen leaving cursor at top left.

START2:LD       A,US
        EMT      OUTC

        RET
```

These files, together with DDT.COM, BASIC.COM, ZASM.COM and CODE.BAS are on the logged-on drive. The file produced by CODE.BAS is to be called SUBROUT.ZSM.

```
A>BASIC CODE

RML Extended BASIC V 5.0L
Copyright (c) 1982 by Research Machines

Output filename ? SUBROUT.ZSM

Segment name ? TEST1
```

(This is the name of the subroutine)

```
Segment entry point ? START1
```

(The segment entry point is the label used within your source file to mark the address to which BASIC should jump when the routine is CALLed)

```
Segment name ? TEST2

Segment entry point ? START2
```

```
Segment name ?                          (No more, so just press RETURN)

Module name ? BASTEST1.ZSM

Module name ? BASTEST2.ZSM

Module name ?                           (No more, so just press RETURN)

Ready: BYE

A>ZASM SUBROUT

ZASM Z80 ASSEMBLER V3.0A
Copyright (c) 1981 by Research Machines

9BC7 bytes free
No errors                               (Hopefully)

A>DDT BASIC.COM
DDT VERS 1.4

NEXT PC
nnnn 0100
-ISUBROUT.HEX
-R
NEXT PC
mmmm 0000
-C

A>SAVE xx NEWBASIC.COM
```

Note: the value xx is obtained by subtracting 1 from the top two digits of hexadecimal number mmmm obtained after invoking DDT, then converting those digits into decimal.

With this example the value mmmm given in DDT was 32F1H so the first two digits are 32H. Subtract 1 from 32H to give 31H. 31(base 16) equals 49(base 10), therefore the value xx in this case is 49.

The file SUBROUT.ZSM should look like this:

```
*H BASIC CODE SUPPORT

USR       EQU     107H          ;USR TRANSFER VECTOR
USRVF     EQU     109H          ;USR VALUE FROM
USRVT     EQU     10CH          ;USR VALUE TO
SAVVAR    EQU     10FH          ;SAVE INTEGER IN VAR
OUTM      EQU     112H          ;PRINT MESSAGE
E.ERR     EQU     115H          ;EXTERNAL ERROR
BBUFV     EQU     118H          ;BASIC BUFFER VECTOR
SUBPTR    EQU     11AH          ;SUBROUTINE POINTER
ENDMEM    EQU     11CH          ;END OF MEMORY
BRFCB     EQU     11EH          ;READ FCB
```

```
BWFCB     EQU   120H        ;WRITE FCB
DEFEXT    EQU   122H        ;DEFAULT EXTENSION
INITSUB   EQU   125H        ;SUBROUTINE INITIALIZATION
CH.INPUT  EQU   128H        ;GET BYTE FROM CHANNEL
CH.OUTPUT EQU   12BH        ;PUT BYTE TO CHANNEL

          ORG   282
          DEFW  CODE0

          ORG   118H
          DEFW  ENDPATCH

          ORG   13010
CODE0:    DEFW  CODE1
          DEFB  5
          DEFM  "TEST1"
          DEFW  START1

CODE1:    DEFW  CODE2
          DEFB  5
          DEFM  "TEST2"
          DEFW  START2

CODE2     EQU   0

*INSERT BASTEST1.ZSM
*INSERT BASTEST2.ZSM

:END OF USER CODE
       DEFB      0
ENDPATCH:
```

Note: The values given above may change with different versions of the BASIC interpreter.

The last example is CAT, which prints a list of the names of the machine-code routines patched into the version of the BASIC interpreter on which CAT is running:

```
 10 DEF FND(A)=PEEK(A)#&100*PEEK(A+1)
 20 LET P=&11A  :REM SUBPTR
 30 LET X=FND(P)                    :REM NODE ADDRESS
 40 IF X=0 THEN 140                 :REM LENGTH
 50 LET P=X
 60 LET L=PEEK(P+2)                 :REM LENGTH
 70 IF L=0 THEN 120
 80 PRINT HEX$(FND(P+L+3)); " " :REM ROUTINE ADDRESS
 90 FOR S=1 TO L
100 PUT PEEK(P+S+2)
110 NEXT S
120 PRINT
130 GOTO 30
140 IF P=&11A THEN PRINT "No CALLs installed"
150 END
```

Other Useful BASEPAGE Addresses

E.ERR This routine displays the message:

```
External error
```

and returns to BASIC. This error (number 1 in the list of BASIC errors) can be trapped by the ON ERROR command in the normal way. A machine-language routine must transfer execution to E.ERR by a jump instruction to generate the error.

OUTM The OUTM routine can be used to output messages to the screen by the normal BASIC output mechanism. Such messages obey current settings set by the WIDTH and NULL commands. If CTRL/P has been entered, output will also go to the printer.

The OUTM routine delivers a message from a table which you must define by tabulating the messages one after the other. The end of each message is indicated by setting (changing to 1) the last bit of the byte representing the last character. This can be done easily if you are using ZASM, the disc-based Z80 assembler (available from Research Machines Limited). Simply insert the up-arrow character in front of the last character of the message:

```
DEFM 'Warnin↑g'
```

This sets the top bit of the "g" byte. In effect, 80 hex is added to its ASCII value.

Since the OUTM routine uses one of the 8 bits to indicate the end of a message, only 7 bits are available for character representation. Hence, it is not possible to output characters with ASCII values greater than 127. Also, ASCII values of less than 32 (usually cursor control characters) cannot be used, except for 10 which gives carriage return/line feed.

You must also define:

- the address of the start of the table (in register HL)
- the index number which points to the message that is to be output (in register E)

An example of the use of the OUTM routine is provided in example 2 of this chapter.

BRFCB This address contains a pointer to the file control block for channel 10.

BWFCB This address contains a pointer to the file control block for channel 10.

CH.INPUT This routine reads a byte from a specified channel. It must be supplied with either:

Register A	—	Channel number to read from
Flags	—	Carry flag set

or:

Register pair IX	—	Pointer to channel I/O block (as returned by a previous call to this routine)
Flags	—	Carry flag in reset state

The second form should be used only after the first form has been used at least once.

The routine returns:

Register A	—	the byte read from the channel
Register IX	—	pointer to I/O block of the channel
Carry flag	—	set for physical end-of-file, else reset

and corrupts registers: AF BC DE HL IX

For example, to read characters repeatedly from a file:

```
        LD      A,10        ;Channel 10
        SCF                 ;Indicate to set up channel
READ:   CALL    CH.INPUT
        .
        .                   ;Deal with character in Register A
        .
        XOR     A           ;Clear carry flag
        JR      READ
```

CH.OUTPUT This routine writes a byte from to specified channel. It must be supplied with either:

Register A	—	Channel number to write to
Register C	—	Byte to write to the file
Flags	—	Carry flag set

or:

Register pair IY	—	Pointer to channel I/O block (as returned by a previous call to this routine)
Register C	—	Byte to write to the file
Flags	—	Carry flag in reset state

The second form should be used only after the first form has been used at least once.

The routine returns:

Register IY — pointer to I/O block of the channel

and corrupts registers: AF BC DE HL IY

For example, to write characters repeatedly to a file:

```
        LD      A,10            ;Channel number
        SCF                     ;Indicate to set up channel
WRITE:  LD      C, <byte to write>
        CALL    CH.OUTPUT
        XOR     A               ;Clear carry flag to
                                ;continued write
        JR      WRITE
```

EXAMPLES

You may wish to try the two examples of machine-language routines and the two useful BASIC programs which are presented in this section.

Example 1

This example allows a printer option to be selected automatically from within BASIC. You could do this with the PRINTER command but, for educational purposes, this example does it with a machine-language routine.

If you want to ensure that a Teletype printer is selected as the system printer before a BASIC program is run, a machine-language routine can be installed in cache memory using the POKE command.

The Teletype is interfaced using a SIO-2 interface and it runs at 110 baud. The COS/ROS Monitor has an Emulator Trap instruction (EMT) called SETLST that selects a printer option automatically. Below is an Assembly Language routine which will call this EMT and select the SIO-2 at 110 baud:

```
0029 =      SETLST   EQU   29H

0000 3E02   SETOPT:  LD    A,2      ;SIO-2
0002 1E00            LD    E,0      ;110 Baud
0004 F729            EMT   SETLST
0006 C9              RET
0000                 END
```

```
Symbols:

SETLST 0029 SETOPT 0000

No errors
```

This loads the A register with 2 to select the SIO-2 port, loads the E register with 0 to select 110 baud, calls COS/ROS with EMT SETLST and returns to BASIC. The routine can run at any address since it contains no address information. The machine code can be extracted (in hex) from the second column:

```
3E 02 1E 00 F7 29 C9
```

The routine can be installed in cache memory using the POKE command; it can be called, by writing a BASIC program such as:

```
10 CLEAR 100,,10 :REM RESERVE SPACE
20 DATA &3E,&02,&1E,&00,&F7,&29,&C9
30 LET EM=PEEK(&11C+56*PEEK(&11D)
40 LET CA=EM+1
50 FOR I=0 TO 6
60 READ V
70 POKE CA+I,V
80 NEXT I
90 CALL CA
```

Example 2

The second example uses the USR and CALL commands. It gives BASIC the ability to store data in sixteen-bit integer arrays.

Some dialects of BASIC have introduced a new data type, the integer numeric variable; this is often declared by following a variable name with a percent sign (for example A9%). Because of the overhead of interpretation, the addition of integer variables does little to speed up execution but provides the advantage of saving space where it is necessary to store a lot of numbers in memory. Integer variables normally occupy only two bytes, as opposed to the normal requirement of four.

The following function and an associated routine allow numbers to be stored in cache memory as if they are in a one-dimensional integer array.

We define the USR function as:

```
LET V=USR(A)
```

where V is the content of integer array element A. The value of A is in the range 0 to N, where 2*(N+1) is the size of cache.

The companion CALL command is:

```
CALL "ST16", V, A
```

which stores the value of V in integer array element A. Note that the range of V is limited to between −32768 and +32767 because only 16 bits are used for storage.

Also included is the equivalent of array bound checking to ensure that only legal values of A are used. If a value of A is outside cache memory, then the error messages:

```
Address out of bounds
External error
```

will be displayed and program execution will stop. This demonstrates the use of the OUTM facility to output a message through BASIC. An Assembly language version of the USR function and CALL command as defined above is:

```
                    ;GENERAL DEFINITIONS

                    ;PATCH is the old content of BBUFV
                    ;Note that any existing machine language
                    ; routine will be overwritten

0006 =              HIMEM   EQU     6
0112 =              OUTM    EQU     112H
0115 =              E.ERR   EQU     115H
0118 =              BBUFV   EQU     118H
011C =              ENDMEM  EQU     11CH
3124 =              PATCH   EQU     3124H
0118                        ORG     BBUFV
0118 8531                   DEFW    BASEND
                    ;--------------

                    ;LINKS TO USR FUNCTION

0106 =              USR     EQU     106H
0109 =              USRVF   EQU     109H
010C =              USRVT   EQU     10CH

0106                        ORG     USR
0106 C32431                 JP      USR16

                    ;LINKS FOR CALL

011A =              SUBPTR  EQU     11AH

011A                        ORG     SUBPTR
011A 3231                   DEFW    CLINK

                    ;---------

                    ;SUBROUTINES PROPER

3124                        ORG     PATCH
```

```
                ;return variable from array

3124 CD0901     USR16:  CALL  USRVF       ;Get arg val into DE
3127 EB                 EX    DE,HL       ;Arg into HL
3128 CD4A31             CALL  CHKCA       ;Check address
312B 46                 LD    B,(HL)      ;Get contents into
312C 23                 INC   HL          ;reg AB
312D 7E                 LD    A,(HL)
312E CD0C01             CALL  USRVT       ;Return as USR value
3131 C9                 RET
                ;---------------

                ;NODE FOR CALL

3132 0000       CLINK:  DEFW  0           ;No more calls
3134 04                 DEFB  4           ;Name length
3135 53543136           DEFM  "ST16"      ;Name of routine
3139 3B31               DEFW  ST16        ;Pointer to routine

                ;STORE VARIABLE IN ARRAY

313B 79         ST16:   LD    A,C         ;Check no of args
313C FE02               CP    2
313E C21501             JP    NZ,E.ERR    ;Error if not 2
3141 E1                 POP   HL          ;Get 2nd arg
3142 CD4A31             CALL  CHKCA       ;Check address
3145 D1                 POP   DE          ;Get 1st arg
3146 73                 LD    (HL),E      ;Store DE at address
3147 23                 INC   HL          ;in HL
3148 72                 LD    (HL),D
3149 C9                 RET               ;Back to BASIC
                ;---------

                ;CHECK CACHE ADDRESS

                ;Convert array subscript to cache offset
                ; Convert offset to address
                ; Check that address is legal

314A 29         CHKCA:  ADD   HL,HL       ;Double subscript
314B DA1501             JP    C,E.ERR     ;Illegal address
314E ED5B1C01           LD    DE,(ENDMEM)
3152 13                 INC   DE          ;Start of cache
3153 19                 ADD   HL,DE       ;Array element address
3154 380E               JR    C,ERROR     ;Illegal address
3156 E5                 PUSH  HL          ;Save address on stack
3157 ED5B0600           LD    DE,(HIMEM)  ;End of free RAM+1
315B 1B                 DEC   DE          ;End of cache
315C B7                 OR    A           ;Clear carry flag
315D ED52               SBC   HL,DE       ;Check upper bound
315F E1                 POP   HL          ;Restore address
3160 D26431             JP    NC,ERROR    ;If illegal address
3163 C9                 RET               ;Else address OK
                ;-----------
```

```
            ; OUTPUT ERROR MESSAGE AND JUMP BACK
            ; TO BASIC.

3164 1E01  ERROR:  LD    E,1        ;1st message in list
3166 216F31        LD    HL,ERRTAB  ;Address of table of errors
3169 CD1201        CALL  OUTM       ;Output error
316C C31501        JP    E.ERR      ;Exit to BASIC with
                                    ;     External Error message

            ;Table of error messages
            ;Each error message must be terminated
            ;with a character with the top bit set.
            ;Putting an up arrow (↑) before the last
            ;character does this.

             ERRTAB:
316F 41646472      DEFM "Address out of bound↑s"

            ;It is vital that the last byte of
            ;BASIC is zero. The following ensures this.

3184 00            DEFB 0

           BASEND:

0000               END

3185 BASEND  0118 BBUFV   314A CHKCA   3132 CLINK   0115 E.ERR
011C ENDMEM  3164 ERROR   316F ERRTAB  0006 HIMEM   0112 OUTM
3124 PATCH   313B ST16    011A SUBPTR  0106 USR     3124 USR16
0109 USRVF   010C USRVT

No errors
```

Note that the values defined here for symbolic constants such as USR and E.ERR, are merely typical. Check the Release Note for the exact values for the current version of BASIC.

The following program tests this routine:

```
10 CLEAR ,,100
20 FOR I=0 TO 9
30 CALL "ST16",I,I
40 NEXT I
50 FOR I=0 TO 9
60 PRINT USR(I)
70 NEXT I
```

APPENDIX A

QUICK REFERENCE GUIDE

This Appendix summarizes the commands, functions, operators and special control characters available in Extended BASIC Version 5 and 6.

COMMAND	PAGE	PURPOSE	
ABS	9.3	Absolute value	
AND	3.6	Logical AND operator	
ASC	9.7	Convert character to number	
ATN	9.2	Arctangent	
ATTRIB	11.5	Return attributes set	
AUTO	4.1	Automatic line numbering	V.6 only
BYE	10.1	Return to monitor	
CALL	18.8	Call machine-language subroutine	
CHR$	9.7	Convert number to character	
CLEAR	6.1	Delete variables, set string, files and cache space	
CLOSE	12.2	Close output file(s)	
CLOSE #	13.2	Close output file	V.6 only
CLOSE INPUT	12.2	Close input file(s)	
CLOSE INPUT #	13.2	Close input file	V.6 only
CONT	4.3	Continue execution after CTRL/Z, STOP or END	
COPY	4.3	Copy a group of program lines	V.6 only
COS	9.1	Cosine	
CREATE	12.3	Open file for output	
DATA	6.2	Define constants	
DEF	9.11	User-defined function or procedure	
DLETE	4.3	Delete program lines	
DIM	6.2	Dimension array	
DIR	4.4	Get disc directory	
EDIT	5.1	Invoke the line editor	
ELSE	8.2	Statement executed if condition is not true	
END	10.1	End of program	
EOF	12.7	Generate end-of-file condition	
EOF #	13.2	Generate end-of-file condition	V.6 only
ERASE	4.4	Erase file	
ERL	8.8	Line number of last error	
ERR	8.6	Error number of last error	
ERROR	8.8	Cause error	
EXCHANGE	6.3	Exchange values of two variables	V.6 only
EXP	9.2	Natural antilogarithm	
FIX$	9.10	Return fixed length string	V.6 only
FLEN	13.10	Return size of a random access file	V.6 only
FN	9.11	User-defined function	
FOR	8.4	Set up loop	

COMMAND	PAGE	PURPOSE	
FPOS	13.10	Return current RA record number	V.6 only
FRE	6.1	Return free memory or string space	
FSAVE	4.5	Save program in internal format	
GET	12.5	Single character input	
GET$	12.5	Single character input	
GRAPH	11.1	Set graph or text mode	
GOSUB	8.1	Call a subroutine	
GOTO	8.1	Transfer control to another line	
HEX$	9.9	Convert to hexadecimal	
IF	8.2	Conditional test	
IMAGE	14.1	Format specification for output	V.6 only
INP	18.4	Input from an I/O port	
INPUT	7.1	Input data from the keyboard	
INPUT #	12.4	Input data from a file	
INPUT LINE	7.2	Input entire line from keyboard	
INPUT LINE #	12.4	Input entire line from file	
INSTR	9.9	Search for a substring	V.6 only
INT	9.2	Truncate to integer	
KILL	10.3	Delete an array	V.6 only
LEFT$	9.8	Extract left portion of a string	
LEN	9.8	Get length of a string	
LET	6.3	Assign to variable	
LINE	11.4	Draw a line	
LIST	4.5	List program on console	
LLIST	4.5	List program on printer	
LLVAR	10.2	List variables on printer	
LNULL	7.5	Set nulls for printer	
LOAD	4.5	Load program from a disc file	
LOAD?	4.6	Check internal format file	
LOADGO	4.6	Load and execute program	
LOCAL	9.15	Define local variables in FN/PROC	V.6 only
LOCAL PRINTER	4.8	Set up printer option	
LOG	9.2	Natural logarithm	
LOOKUP	12.3	Test whether file exists	
LPOS	7.5	Return current position of printer head	
LPRINT	7.4	Output to printer	
LTRACE	10.3	Set line number trace on printer	
LVAR	10.2	Print variables on console	
LWIDTH	7.5	Set width of printer	
MAX	9.4	Return the maximum of two values	V.6 only
MERGE	4.7	Merge programs	
MERGEGO	4.7	Merge and execute programs	
MID$	9.8	Return middle of a string	
MIN	9.4	Return the minimum of two values	V.6 only
MOD	9.4	Return a modulus value	V.6 only
NETWORK PRINTER	4.8	Set up printer option	
NEW	4.8	Clear all program statements and variables	
NEXT	8.4	Return to beginning of a loop	
NOT	3.6	Logical NOT operator	

COMMAND	PAGE	PURPOSE	
NULL	7.5	Set nulls for console	
ON	8.3	Indexed transfer of control	
ON BREAK	8.8	Trap console interrupt	
ON EOF	12.7	Trap end-of-file	
ON EOF #	13.2	Trap end-of-file	V.6 only
ON ERROR	8.6	Trap error condition	
OPEN	12.3	Open file for reading	
OR	3.6	Logical OR operator	
OUT	18.5	Output to an I/O port	
PEEK	18.4	Return data from a memory location	
PLOT	11.2	Plot string, character, or dot	
POINT	11.5	Test screen location	
POINTS	11.5	Test screen locations	
POKE	18.4	Change a memory location	
POS	7.4	Return current console cursor position	
PRINT	7.3	Output to console	
PRINT #	12.6	Output to a file	
PRINTER	4.8	Set up printer option	
PROC	9.19	User-defined procedure	V.6 only
PUT	12.6	Single character output	
QUOTE	12.8	Set string quotes on output	
RANDOM	13.4	Create, or open, a random access file	V.6 only
RANDOMIZE	10.3	Change seed used by random number generator	
READ	6.4	Move data from a DATA statement to a variable	
READ #	13.5	Read from a random access file	V.6 only
REM	10.1	Remark or comment	
RENAME	4.9	Rename file	
RENUMBER	4.9	Renumber program and change line number references	
RESET	4.10	Initialize discs	
RESTORE	6.4	Reset data pointer	
RESUME	8.6	Resume execution after error	
RETURN	8.1	Return control from a subroutine	
RIGHT$	9.8	Return right portion of a string	
RLEN	13.11	Return record length of RA file	V.6 only
RND	9.3	Random number	
RPOS	13.11	Space remaining in a record	V.6 only
RUN	4.11	Start execution of program	
SAVE	4.10	Write a copy of the program to a disc file	
SGN	9.3	Get sign of expression	
SIN	9.1	Sine	
SPACE$	9.10	Returns a string of spaces	V.6 only
SPC	7.4	Print spaces	
SQR	9.2	Square root	
STEP	8.4	Set FOR increment	
STOP	10.1	Terminate program execution	
STR$	9.8	Convert value to string	
STRING$	9.11	Returns a repeated string	V.6 only
TAB	7.4	Tabulate to column	

COMMAND	PAGE	PURPOSE	
TAN	9.1	Tangent	
TEXT	11.2	Set text mode	
THEN	8.2	Statement executed if condition is true	
TO	8.4	Set upper limit in FOR statement	
TRACE	10.2	Set line number trace	
TYP	13.9	Return data type from RA file	V.6 only
USING	14.1	Format specification for output	V.6 only
USR	18.7	User provided machine-code routine	
VAL	9.7	Convert string to number	
VARADR	18.6	Get variable address	
WAIT	18.5	Wait for input status bit	
WIDTH	7.5	Set width of console	
WRITE #	13.7	Write to a random access file	V.6 only
XOR	9.5	Exclusive OR of two values	V.6 only
?	7.4	Equivalent to PRINT	
'	10.1	" ' " form of comment	V.6 only
command.	4.2	" . " form of line number	V.6 only

HRG CALLS

CALL	PAGE	PURPOSE
CLEAR	16.2	Clear current page and view
COLOUR	16.3	Set the actual intensity or colour displayed that corresponds to the specified logical intensity
DISPLAY	16.5	Specify the page and view to be displayed
FILL	16.6	Fill the given rectangle
GREAD	16.7	Load graphics memory from disc file
GWRITE	16.8	Save graphics memory in disc file
LINE	16.9	Draw a line from the current position to given point
OFFSET	16.10	Change coordinates of bottom left corner of screen
PLOT	16.11	Plot a point on the screen
RESOLUTION	16.12	Select high, medium or extra high resolution and set number of bits/pixel
SETCOL	16.13	Modify values held in memory-copy of colour look-up table without affecting the table itself
UPDATE	16.14	To make a particular page and view accessible for modification by PLOT, LINE or FILL
VIEW	16.15	Transfer colour changes specified by SETCOL to colour look-up table

HRG CALLS (Level 2 only)

CALL	PAGE	PURPOSE
CHARSIZE	17.3	Define magnification of plotted characters
COPY	17.4	Make a copy of a rectangle
DEFCHAR	17.5	Define characters for use by STPLOT
DUMP	17.7	Copy a rectangle from the screen to the printer
PATSIZE	17.9	Define size of point printed from the screen
PRINTER	17.13	Define printer type and print density
RDOUT	17.14	Read the logical intensity of a point
SHADING	17.15	Define shading pattern
STPLOT	17.16	Plot a character string

PREDEFINED VARIABLES

NAME	PAGE	VALUE	
EE	9.6	2.71828	V.6 only
PI	9.6	3.14159	V.6 only

OPERATORS

These operators are arranged in decreasing order of precedence. Operators of equal precedence are evaluated from left to right.

+	String concatenation	
↑	Exponentiation	
—	Negation	
* /	Multiplication, division	
MOD	Modulus Arithmetic	V.6 only
+ —	Addition, subtraction	
MIN	Minimum	V.6 only
MAX	Maximum	V.6 only
< > = <> >= <=	Relational operators	
NOT	Logical negation	
AND	Logical AND	
OR	Logical OR	
XOR	Logical Exclusive OR	V.6 only

CONTROL CHARACTERS

The control characters marked * act as toggles on the feature they control. Pressing once activates the feature, which persists until they are typed again:

CTRL/@ *	Toggle smooth scrolling (380Z, COS 4.0, 80-char. only)
CTRL/A *	Toggle autopaging
CTRL/C	Return to CP/M (after check)
CTRL/E *	Echo console output to printer
CTRL/F	Enter Front Panel (after check)
CTRL/L	Clear screen, cursor bottom left
CTRL/_	Clear screen, cursor top left
CTRL/M	Terminate Line
CTRL/P *	Echo console output to printer
CTRL/Q	Resume execution after CTRL/S
CTRL/S	Suspend program execution
CTRL/U	Delete current line
CTRL/Z	Interrupt execution of program, erase line
DELT	Delete previous character

APPENDIX B

ERROR MESSAGES

This Appendix gives a list of error messages and their causes. When an error occurs in a program, the error message will be followed by "at line n", where n is the number of the statement BASIC was executing when the error was detected. For example:

```
10 PTINT "HELLO"
RUN

Syntax error at line 10
Ready:
```

The numbers given with each error are the error numbers used in conjunction with the ERROR command and the ERR function.

1 External error

This error can be generated by machine-code routines, but should not occur otherwise in the absence of ERROR commands.

2 NEXT without FOR

A NEXT statement has been encountered without a corresponding FOR statement. This error can also occur if GOSUBs and FOR loops are badly nested.

3 Syntax error

This usually means that an unexpected character has been found, suggesting an invalid statement. This is the most common error message.

4 RETURN without GOSUB

A RETURN statement has been encountered without a corresponding GOSUB.

5 Out of data

There are insufficient data items in the program to satisfy all of the READ commands.

6 Illegal function

This is generated when a command is presented with illegal arguments. This usually means that the arguments are not in the expected range.

7 Arithmetic overflow

The result of a calculation lies outside the permitted range, i.e. the magnitude of the number is greater than about 1E38. This error can also occur if ridiculous numeric constants are provided, for example in DIM statements.

8 Out of memory

There is insufficient memory available to store the program and variables. The most common cause is that arrays are DIMensioned too large for the available memory.

9 Undefined statement

A statement number, after GOTO, GOSUB, RESTORE, etc., references a line which does not exist.

10 Subscript out of range

A subscript of an array is negative or greater than the upper bound of the corresponding dimension as declared in the DIM statement. This can also occur in a DIM statement if the memory size is greatly exceeded.

11 Redimensioned array

An array in a DIM statement has already been DIMensioned.

12 Can't divide by zero

This error is caused by an attempt to divide by zero.

13 Illegal direct

Some commands, such as INPUT and DEF, are illegal in direct mode, i.e. without a line number. Attempting such a command results in this message.

14 Type mismatch

A numeric expression was found where a string expression was expected, or vice versa.

15 No string space

BASIC needs more space in which to store strings than is available in string space. String space should be increased with CLEAR.

16 String too long

An attempt has been made to create a string longer than 255 characters.

17 String too complex

A string expression is too complicated for BASIC to handle. This usually means that string functions are too deeply nested. The expression should be split into two or more parts using intermediate variables.

18 Can't continue

The CONT command has been used in an illegal context. It can only be used after interrupting a running program with <CTRL/Z>, STOP or END and before modifying the program.

19 Undefined user call

A user-defined function (after FN) has been referenced without having first been defined. This error is unlikely to occur in Extended BASIC Version 6, as the function does not have to be defined before it is referenced.

20 Illegal EOF

The end of the input file has been reached, and there is no ON EOF currently active.

21 Files different

The file inspected with LOAD? is not the same as the program held in memory.

22 Recovered

This message is generated after BASIC is entered via the restart address (given in the Release Note). It should not occur in normal operation.

23 Name not found

The subroutine name in a CALL command cannot be found in the subroutine name list.

24 Can't verify ASCII files

A LOAD? command has been presented with an ASCII file — i.e. the file had been SAVEd, not FSAVEd. BASIC cannot do this.

25 Can't MERGE internal files

A MERGE command has been presented with an internal format file.

26 Unknown error

This error is no longer defined.

27 Read error

An error has occurred during a read operation from file. If this occurs there is probably a hardware fault. This can also occur on a network system if an attempt is made to read from a disc that has been removed.

28 RESUME without error

A RESUME command has been encountered but no error has occurred.

29 Record too long

A line in an input data file is longer than the input buffer, which is 130 characters in Extended BASIC Version 5, and 253 characters in Extended BASIC Version 6.

30 Invalid unit number

An invalid channel specification has been given. The only channels allowed are 0, 2 and 10 (and 20-127 in BASIC 6 only), and some operations, such as OPEN and CREATE, are not allowed on channels 0 or 2. This error is also the result of trying to INPUT from the printer, or, in BASIC 6, trying to read from a file open for writing or vice versa.

31 Missing file name

The file specification after LOAD, SAVE, or MERGE is missing.

32 No input file

An attempt has been made to read from an input file (with INPUT or INPUT LINE) without first having OPENed it.

33 No output file

An attempt has been made to write to an output file (with PRINT, DIR, etc.) without first having CREATEd it.

34 Invalid device

The device name part of a file specification is illegal, i.e. it is not one of CON:, RDR:, PUN:, LST:, or A: to P:.

35 Invalid file name

The file name part of a file specification is illegal.

36 Write error

An error has occurred during a write operation to file. This could occur on disc systems if the disc becomes full, but usually implies a hardware malfunction. It should be very rare. On a network system this can occur if a permanent disc error (such as the disc not being present) is detected during a write operation.

37 Wrong internal format

A file presented to LOAD or LOADGO is in an internal format incompatible with this BASIC. Either the file is corrupt or it was prepared from another version of BASIC.

38 File not found

The file specified after LOAD, MERGE, OPEN, or RENAME could not be found. On cassette systems this will be generated only on the ROM Pack channel.

39 Directory full

The disc directory already holds the maximum number of files allowed and an attempt has been made to add another. An extent to a file is treated as a separate file, so this error could occur after any disc write operation. This error will not occur on cassette sytems.

40 No disc space

The disc is almost full. There will just be sufficient room to CLOSE the output file, but to proceed you must either erase something or change the disc and issue a RESET command. This error will not occur on cassette systems.

41 Close error

An error has occurred during a CLOSE command, and BASIC was unable to CLOSE the file. This error is usually caused by attempting to CLOSE a file which has been ERASEd or RENAMEd. This error will not occur on cassette systems.

42 ROM Pack read error

An error has occurred during a ROM Pack operation. Usual causes are the ROM Pack being removed while being read, an integrated circuit in the ROM Pack not inserted correctly or a faulty ROM Pack. This error can be generated only on a 480Z cassette system.

43 Read only file/disc

An attempt has been made on a network system to write to a read-only file or disc. If the disc is not physically write-protected, and no other user has any files open on that drive, then RESET will set the disc to read/write status.

44 File in use

An attempt has been made on a network system to open a file that another user has open for writing.

45 Too many files on Network

Either you have attempted to open more than 8 files, or the file limit on the network server has been reached.

46 Not logged in

This indicates that the network server has been restarted without CP/NOS being reloaded into the station.

47 Unknown I/O error

This means BASIC has received an unexpected error from a network. This should not normally occur and must be reported to the network manager if it does.

48 No end to DEF FN/PROC

This error occurs if a function or procedure definition contains another function or procedure definition, or if there is no FNEND or PROCEND statement to complement a DEF statement. It may also occur during checks that take place following LOAD or RUN commands.

49 No start to FN/PROC END

This error occurs if an FNEND, PROCEND, FNRETURN or PROCRETURN command is encountered outside a function or procedure definition. It may also occur during checks that take place following LOAD or RUN commands.

50 Illegal FN/PROC exit

This means that either a PROCEND/RETURN is followed by an expression, an FNEND/RETURN is not followed by an expression, FNEND/RETURN is being used to exit from a procedure, or PROCEND/RETURN is being used to exit from a function.

The last two could result from badly-programmed GOTO statements.

51 Format error

There is an error in a format specification. The format string is displayed, with an up-arrow indicating the position at which the error was found.

52 Too many files

This error occurs if an attempt is made to open a larger number of files than declared in a CLEAR statement. The default value is 2 files. Remember that LOAD and SAVE require a file channel.

53 Invalid record length

This error is given if you try to open a random access file with a different record length from that with which it was created, or with a record length of less than 3 bytes.

54 Invalid file type

This error is given if you try to open a sequential file for random access, or vice versa.

A common cause of this error is the restarting of a program that has stopped, if the program uses a random access file and that file was not closed. (See Chapter 13 for further details).

55 Reading unwritten data

This error is given if the data being read from a random access file is of an unrecognizable type, usually due to reading an unwritten record.

56 Record number too large

This error is given if an attempt is made to write to a random access file a record with a record number greater than 65535, or to read from a random access file a record with a record number greater than the highest so far written to that file.

57 File open for reading only

This error occurs if you try to write to a random access file that has been opened in READ mode.

58 Invalid function for CP/M 1.4

This error occurs if an attempt is made to use the RANDOM command when running under version 1.4 of the CP/M operating system. The random access facilities are not available in that version.

59 File locked

This results from an attempt to access a *locked* file with READ, WRITE, CLOSE, CLOSE INPUT or TYP.

Unknown error

This error is generated if BASIC attempts to produce an error message not in its list. It implies either corruption of the interpreter or an ERROR command has been executed with an argument outside the permitted range.

Interrupted

This message, while not strictly an error message, is included here for convenience. It is generated when BASIC executes a STOP command or when program execution is interrupted by pressing <CTRL/Z>. In either case program execution can be resumed by the CONT command.

***Invalid input**

This message is produced when a response to an INPUT statement requiring a number starts with a string. The entire line must be repeated. This error does not terminate the program.

***Extra lost**

The response to an INPUT statement contained more data items than were required to satisfy it. The excess items are lost. This error does not terminate the program.

APPENDIX C
CHANGES FROM PREVIOUS VERSIONS

This Appendix describes first most of the differences between Extended BASIC version 5 and DBAS9 version 3.0. Changes from version 4 are described next, followed by changes made to version 5 since version 5.0 A, and finally a summary is given of the differences between the current versions 5 and 6.

CHANGES FROM DBAS9

Because a number of keywords have been added, the choice of variable names is rather more restricted in Extended BASIC version 5 than in DBAS9 or version 4, as variables must not contain embedded keywords.

The WIDTH command has been changed to allow an infinite line length, which is now the default. It now accepts a channel number after #, and the WIDTHs for the console, printer, and file channels are all independent.

The NULL command can now accept a channel number after #, and all three output channels have independent NULL settings.

The QUOTE command has been introduced to simplify file handling. The default situation is unchanged.

The random number generator is initialized by typing RUN. This means that each RUN of a program will get the same sequence of randoms unless the RANDOMIZE command is issued.

ON EOF now works in a slightly different manner. It traps the end of file situation directly, instead of checking just at the ON EOF statement itself. This means that the ON EOF command need not be placed within the INPUT loop, and it works with multiple data items per line. The form ON EOF disables end-of-file checking.

The EOF command causes BASIC to react as if the end-of-file had been reached.

All of the commands which produce output can now be followed by a channel specification to redirect the output.

A string entered at the keyboard in response to an INPUT command need no longer be enclosed in quotes if it contains a colon. If an INPUT statement includes a prompt terminated with a comma, the normal question mark prompt is suppressed.

The INPUT LINE command has been added as an enhancement of INPUT, allowing all of the input to be transferred to the variable including spaces and commas without the need to enclose the date in quotation marks. LINE INPUT is equivalent to INPUT LINE.

Numeric constants may now be supplied in hexadecimal, preceded by an ampersand (&). The CLEAR command can now take a second argument, which specifies cache memory. The default string space size is now 100 bytes.

RESTORE can now take an optional line number, which allows the data pointer to be set to the specified line.

The BYE command has been added, chiefly to allow a program to leave BASIC. It first CLOSEs any files.

The PLOT command has been extended. It no longer reports an error if the point is off the screen. If the third argument is a string, it is PLOTted directly, and if it is missing, BASIC uses the last number given. With the 380Z a fourth argument to PLOT sets attributes on an 80-character board.

The LINE command has been added to increase the speed of line drawing.

The functions POINT and POINTS have been added, which return the contents of the screen. With the 380Z 80-character board the ATTRIB function returns any set attributes.

The CALL command has been added, which provides far greater flexibility in calling machine-language subroutines than is possible with USR.

The function VARADR has been added, which returns the address of a variable. This is of use chiefly with machine-language subroutines.

The LOADM command becomes MERGE. The effect is unchanged. The FSAVE command allows saving of internal format program files, which can be loaded extremely rapidly. LOAD deals with this type of file as well. The commands LOADGO, MERGEGO, and LOAD? have been added.

All of the routines accessible by the FILES command can now be called by more mnemonic names, viz.

FILES 0	RESET
FILES 1	OPEN
FILES 2	CREATE
FILES 3	CLOSE
FILES 5	ERASE
FILES 6	RENAME

In addition, the DIR command has been added, and can if desired be accessed by FILES 7.

The LOOKUP function has been added to allow a program to determine whether or not a file exists.

The OPEN, CREATE, and CLOSE commands should be followed by a channel number, although in this version no error is caused if it is omitted.

An attempt to CREATE a file which already exists no longer causes an error. Instead, the message:

```
File exists--replace(Y/N):
```

is output and action taken according to the response. The ERASE command no longer reports an error if the file did not exist.

The forms PRINT #; and INPUT #; have now been replaced by PRINT #10 and INPUT #10, although the semicolon form still works.

The commands GET and GET$, for single character timed input, and PUT for single character output, have been added.

Within 380Z EDIT, the R (replace) command can be aborted by the ESC or RETURN keys in the same manner as Insert mode. The K command no longer echoes the deleted characters between backslashes.

The DELETE command no longer fails if the final line does not exist; it will now DELETE the same lines that LIST would list.

The ON ERROR command has been added, which with the RESUME command and the functions ERR and ERL allows program control of execution errors. The ERROR command causes an error.

The ON BREAK command traps console interrupts with <CTRL/Z>.

BASIC stores the last key struck while a program is running. This effectively allows one key type ahead. It also allows <CTRL/Z> to interrupt a program while the screen is paging.

It is now possible to CONTinue after an INPUT statement has been interrupted with <CTRL/Z>.

<CTRL/E> and <CTRL/P> now both cause console output to be echoed to the printer. <CTRL/S> stops the program running. <CTRL/Q> resumes execution and <CTRL/Z> interrupts the program.

The PRINTER command has been included to allow selection of a printer directly from BASIC, without entering the Front Panel.

The HEX$ function has been added.

CHANGES FROM VERSION 4

This section describes the differences between version 5 and version 4 of Extended BASIC.

The GET and GET$ functions and the PUT command have been added.

The FSAVE command has been added, and the LOAD command modified to cope with the internal format files it produces. The LOAD?, LOADGO, and MERGEGO commands have been added.

ON ERROR, RESUME, ERROR, ERR, and ERL have been added.

The ON BREAK command has been added.

The PLOT command has been greatly extended. The LINE command and the POINT, POINTS, and ATTRIB functions have been added.

The PRINTER command has been added.

The default quote character for the file device has been changed back to zero, maintaining compatibility between DBAS9 and version 5.

The question mark prompt on an INPUT or INPUT LINE statement is suppressed if a supplied prompt is terminated by a comma.

<CTRL/O> no longer suspends program output.

CHANGES FROM VERSION 5.0 A

Lower case letters are now recognized as forming part of numbers where appropriate, i.e. "e" in exponents and "a" to "f" in hexadecimal numbers.

A check is made that a line number after GOTO, GOSUB, THEN etc. terminates the statement. Previously, no check was made.

Any ON ERROR flag is cleared when BASIC is reentered via its restart address (normally 103 hex). This means that the program does not automatically resume execution.

Line 0 (zero) now never permitted.

The warning messages "*Extra lost" and "*Invalid input" do not appear after INPUT from a file. Instead, these error conditions are ignored, with a new line being read after "*Invalid input".

The LINE INPUT statement has been added and is identical to INPUT LINE.

Error 20, "Illegal EOF", becomes "Illegal end-of-file".

Lines not starting with a digit are ignored on LOAD, LOADGO, MERGE, and MERGEGO. This allows correct parts of corrupt files to be loaded successfully. As a result, the error message "Missing statement number" disappears. The statement ERROR 26, which would have generated this error, now results in "Unknown error".

Typing <CTRL/O> now has no effect, instead of suppressing program output.

When <CTRL/F> is pressed, the message:

```
↑F--are you sure (Y/N):
```

appears and BASIC waits for a key to be typed. If the response is Y or y the Front Panel is entered. Otherwise BASIC continues execution. If <CTRL/F> is typed while GET or GET$ is waiting for a character, the value 6 is returned and the Front Panel is NOT entered.

DIFFERENCES BETWEEN VERSIONS 5 AND 6

In Version 6 the "FILES n" group of commands has been removed altogether.

All other features listed in this section are those provided by version 6 that are not included in version 5.

Commands	AUTO
	CLEAR (specifying number of files)
	CLOSE #
	CLOSE INPUT #
	COPY
	EOF #
	EXCHANGE
	KILL
	ON EOF #
	RANDOM
	READ #
	RENUMBER (with a third argument)
	RESUME (with line number or NEXT)
	WRITE #
Functions	FIX$
	FLEN
	FPOS
	INSTR
	MID$ (on left of "=")
	RLEN
	SPACE$
	STRING$
	TYP

Operators	MAX MIN MOD XOR
" . " equivalent to last line number referred to	
" ' " form of comment	
Error codes above 47	
Pre-defined variables	EE PI
Formatted output	PRINT USING CREATE USING IMAGE
User-defined functions	— flow of execution — number of arguments — string arguments — return of string value — multiline user-defined functions
User-defined procedures	
Simultaneous use of many more files (extra channel numbers)	

APPENDIX D

SELECTING A PRINTER

SETTING UP A PRINTER

Connecting a printer to the computer requires that the printer be plugged into an appropriate connector on the computer rear panel, and informing the computer that you have done this. Both of these operations, while being simple in principle, are very easy to get wrong, so if it doesn't work, try again after rereading the instructions.

If your printer has a parallel interface, it should be plugged into the User I/O socket on the back panel. This is a 25-way D submin female socket connected to the boards of the computer by a flat 20-way cable. It may be marked P1.

There are three types of interface supplied with the 480Z, one parallel and two serial. The serial interfaces are 8-way DIN sockets.

There are several types of serial interface supplied with the 380Z. Most of them appear on the back panel as 25-way D submin sockets, connected to the boards by relatively few wires — 10 at most and probably less. It may be marked S1. 20 mA current loop interfaces use a 6-pin DIN connector.

If you are unsure of the type of interface on your printer, check the socket on the printer. All parallel type printers sold by Research Machines accept a plug which has a single oblong plastic tongue surrounded by 36 contacts. No serial printers use this sort of connector.

After connecting your printer to the computer you must set up a printer option. In BASIC, this is normally done by the PRINTER command. The first number supplied indicates the type of the printer. Parallel interfaces (e.g. Anadex, Centronics) are always type 3. Printer type 0 is the screen, which means that all output sent to the printer (e.g. with LPRINT) appears on the screen. The computer normally comes on in this state.

On a 480Z, SIO-2 serial interfaces are printer type 2. Type 4 is used for SIO-4 interfaces.

On a 380Z, SIO-2 and SI0-2B serial interfaces are printer type 2. Type 4 is used for SIO-4 interfaces. Type 1 is for SIO-1B and SIO-1A interfaces. Printer types 5 and 6 are used for an SIO-5 or SIO-6 respectively.

If you are unsure of the particular type of your 380Z serial interface, check the board inside the computer which connects directly to the socket on the back panel. If the board is a small one, about three inches by two, it is either an SIO-2 or SIO-2B, and therefore type 2. If it is a standard sized card (about 10"×5") with about a third of the board completely blank, it is a type 1 interface. Otherwise it is type 4, 5 or 6.

Printer types 2, 4, 5, and 6 for the 380Z and printer types 2 and 4 for the 480Z require a "Baud code", a number which indicates the Baud rate of your printer. This is a measure of how fast the printer is, and the rate at which the computer sends information to be printed obviously must match the rate at which the printer expects to receive. Check the printer to find out the expected Baud rate. Then select the appropriate code from the table below, and pass this as the second argument to PRINTER.

Code	Baud rate
0	110
1	300
2	600
3	1200
4	2400
5	4800
6	9600

Printer type 2 will not go faster than 2400 Baud. With a 380Z printer, type 1 baud rates are set by a small switch on the rear panel.

After a printer has been selected, test the printer by trying

```
LPRINT "HELLO, WORLD"
```

If this prints nothing, then check the printer type and that you have connected the printer correctly. If it prints garbage, then the Baud rate is probably set incorrectly; try another. If the message appears on the screen, then printer type 0 has been selected. This often means that the type number was too large.

EXAMPLES

For an Anadex DP-9501 printer with a parallel interface, use: `PRINTER 3`

For a Teletype with an SIO-2 serial interface, use: `PRINTER 2,0`

For a Qume daisywheel with an SIO-4 serial interface, use: `PRINTER 4,3`

For an EPSON printer with a parallel interface, use: `PRINTER 3`

For an EPSON printer with a serial interface, use: `PRINTER 4,6`

APPENDIX E

COLOUR LOOKUP TABLE

The elements of the colour lookup table that apply in the various modes are tabulated in this Appendix. These tables should be referred to when using SETCOL. In the three medium resolution modes, all 16 values must be set up by SETCOL before calling VIEW. In the two high resolution modes only the first four values need be defined.

HIGH RESOLUTION, 2 BITS/PIXEL (HR2)

The first 4 elements of the colour lookup table correspond directly to intensities 0 to 3.

Table Address	View 0 Intensity
0	0
1	1
2	2
3	3

Values assigned to table addresses 4 to 15 will be ignored.

HIGH RESOLUTION, 1 BIT/PIXEL (HR1)

The first 4 elements of the colour lookup table govern 2 views as follows:

Table Address	View 1 Intensity	View 0 Intensity
0	0	0
1	0	1
2	1	0
3	1	1

Values assigned to table addresses 4 to 15 will be ignored.

MEDIUM RESOLUTION, 4 BITS/PIXEL (MR4)

The 16 elements of the colour lookup table correspond directly to intensities 0 to 15

MEDIUM RESOLUTION, 2 BITS/PIXEL (MR2)

The 16 elements of the colour lookup table govern 2 views as follows:

Table Address	View 1 Intensity	View 0 Intensity
0	0	0
1	0	1
2	0	2
3	0	3
4	1	0
5	1	1
6	1	2
7	1	3
8	2	0
9	2	1
10	2	2
11	2	3
12	3	0
13	3	1
14	3	2
15	3	3

Thus, to see those pixels that are of intensity 3 and are common to views 0 and 1, set element 15 to be visible, and the rest to background.

MEDIUM RESOLUTION, 1 BIT/PIXEL (MR1)

The 16 elements of the colour lookup table govern 4 views as follows:

Table Address	View 3 Intensity	View 2 Intensity	View 1 Intensity	View 0 Intensity
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

Thus, to see those pixels that are common to views 2 and 3, set elements 12 to 15 to be visible, and the rest to background.

APPENDIX F

ESCAPE SEQUENCES

An escape sequence is a character string sent to the screen and recognized as special because of certain characteristic features. Instead of the characters appearing on the screen, other actions take place.

Escape sequences may be generated within a BASIC program by using PUT statements, and an example of their use would be to switch a screen capable of operating in either 40- or 80-character mode into the mode required for the current program.

Escape sequences can only be used on 480Z and 80-character 380Z systems. On a 40-character 380Z the characters are output literally, and, because some of the values will have undesireable effects, they are probably best avoided.

CHARACTERISTIC FEATURES

An escape sequence is distinguishable from any other string of characters by the following characteristics:

- Its first character is the ESCAPE character (ASCII 27)
- The second character of an escape sequence is known as the Sequence Introducer (SI), which must be one of the characters from the list below, and its value determines the number of subsequent characters which are assumed to be part of the escape sequence
- The subsequent characters needed for each SI are also shown below, and may include a Switch (SW) and/or a Control Parameter (CP)
- A Switch may take the value "0" (to switch a feature off) or "1" (to switch it on)
- A Control Parameter may take any value given in the lists below.

ERRORS IN ESCAPE SEQUENCES

Once an ESCAPE character has been sent to the screen to denote the beginning of an escape sequence and if the SI is not recognized then the escape sequence is immediately terminated and any further characters in the sequence are displayed on the screen in the usual way.

Illegal characters after a legal SI will invalidate the sequence but will not affect the number of characters diverted from the screen.

Any escape sequence in which an error is detected is not executed.

ESCAPE SEQUENCES

1. Send a special character to the screen:

SI = "!" (ASCII 33)

The sequence after the SI is a single byte to be output directly to the screen, with the characters in the range ASCII 0-31 or ASCII 127 *not* being interpreted as special characters.

```
Example:          PUT 27,"!",10
or, if preferred, PUT 27,33,10
```

sends a character with the ASCII value 10 directly to the screen.

2. Set switches on or off:

SI = "=" (ASCII 61)

The sequence after the SI is a switch (SW) followed by a CP.

CP = "B" Switch off or on the underline attribute

CP = "C" Switch off or on the dim attribute

CP = "D" Switch off or on the reverse video attribute

CP = "F" Switch off or on the <CTRL/F action in CP/M and COS/ROS (not the <CTRL/F> action in BASIC)

CP = "G" Switch off or on the <CTRL/A> action (See "Controlling Screen Output" in Chapter 2)

CP = "I" Switch off or on the text in HRG output

CP = "J" Switch off or on the 80-character mode (only available on an 80-character system)

CP = "L" Switch off or on the alternate character set

Example:

```
PUT 27,"=1J"
```

will ensure that the 380Z/480Z is operating in 80-character mode (if it is capable of doing so).

Note: If the screen is in "graphics" mode when you set the J switch, the screen will return to "text" mode.

3. Restore the function keypad keys (480Z only) to their original uses:

SI = ">" (ASCII 62)

The sequence after the SI is a CP only.

CP = "D" Restore all of the function and arrow keys to their original values (i.e. undo the effects of SI = "%" described in 6, below). Note that this does not redefine them to the values set by BASIC for its editor.

Example:

```
PUT 27,">D"
```

4. Define or redefine the display scrolling window:

SI = "?" (ASCII 63)

The sequence after the SI is 4 bytes followed by a CP.

CP = "A" The 4 bytes define a scrolling window, in the order X(lower), X(upper), Y(lower), Y(upper), and the values must be such that:

0 <= XL <= XU <= 39 (40-character width screen), or
0 <= XL <= XU <= 79 (80-character width screen),

and:

0 <= YL <= YU <= 23.

CP = "B" The 4 bytes define a rectangular area to be cleared, with order and value limits as above.

Example:

```
PUT 27,"?",0,39,20,23,"A"
```

will define a scrolling window to be the full width of a 40- character screen, using only the bottom 4 lines of the screen.

5. Beeper sound:

SI = "@" (ASCII 64)

The sequence after the SI is 2 bytes followed by a CP.

CP = "A" The 2 bytes define the frequency and duration respectively of the beeper (which is sounded by <CTRL/G>).

Example:

```
PUT 27,"@",150,75,"A"
```

Note that only 480Z systems have beepers fitted as standard.

6. Define new uses of the function keypad keys (480Z only):

SI = "%" (ASCII 37)

The sequence after the SI is a CP, followed by a byte (of value n), then n further bytes. The sequence is used to redefine the character string that is generated when a function or arrow key is pressed. The new character string will be the last n bytes of the escape sequence.

The character string can be of any length from 0 to 127 bytes, but the total number of characters for all keys is also limited to 127. If the string is too long the escape sequence will have no effect.

CP = "A" Redefine <up-arrow>
CP = "B" Redefine <right-arrow>
CP = "C" Redefine <down-arrow>
CP = "D" Redefine <left-arrow>
CP = "E" Redefine <F1>
CP = "F" Redefine <F2>
CP = "G" Redefine <F3>
CP = "H" Redefine <F4>
CP = "I" Redefine <SHIFT/up-arrow>
CP = "J" Redefine <SHIFT/right-arrow>
CP = "K" Redefine <SHIFT/down-arrow>
CP = "L" Redefine <SHIFT/left-arrow>
CP = "M" Redefine <SHIFT/F1>
CP = "N" Redefine <SHIFT/F2>
CP = "O" Redefine <SHIFT/F3>
CP = "P" Redefine <SHIFT/F4>

Example:

```
PUT 27,"%G",4,"RUN",13
```

will redefine the F3 function key so that it generates the string:

```
RUN<RETURN>
```

7. Define a dot pattern:

SI = "<" (ASCII 60)

The sequence after the SI consists of the 13 bytes:

c, b1, b2, b3, b4, b5, b6, b7, b8, b9, b10, b11, b12

where c is the character (in the range ASCII 128 to 255) that is to have its dot pattern in the Writable Character Store redefined, and b1, b2, ... ,b12 contain the new dot pattern.

Index

USER'S COMMENTS

To help Research Machines to produce the highest quality microcomputers, supporting software, and technical publications, we like to hear from users about their experiences with our products.

Do share your thoughts with us by jotting them down on the tear-off form on the next page. You can leave out your personal details, if you want to. Fold the form in two, seal it with a piece of adhesive tape, and put it in the post. No stamp is needed if you post it within the United Kingdom.

If you would like to give more information than we have allowed room for on the form, we will be very pleased to receive a separate letter from you. You can even use the form to ask for a post-paid envelope, if you wish.

Additional information will be most useful, if you give us as much detail as possible about your hardware configuration, software version number, or manual title, so that we can relate your comments to the correct products.