ENTERPRISE

AUTO
GRAPHICS
TEXT
FUNCTION 7
STATUS
HELP
FUNCTION 8
ENTERPRISE
SIXTYFOUR
HOLD
ERASE
STOP
DEL
ENTER
INS
ALT

PROGRAMMING GUIDE

Version 2

CONTENTS

GETTING ACQUAINTED

INTRODUCTION

Let's begin by using the machine. That way you can get used to it and see demonstrations of its abilities. Technicalities will be explained later.

It is best if you follow this part of the manual in sequential order, because it is designed to help you to get acquainted with your new computer. Wherever there is scope for you to follow up one subject before you go on to look at others, cross references are provided, so you can learn about the machine in whatever order and at whatever speed you feel comfortable. The second part covers each 'section' of programming (they all link up in reality as you will see) in detail and on a slightly higher level than in the first part. The Reference Section at the end will help you to discover more about the Enterprise once you have learned the fundamentals of controlling the computer.

Before you look at some programs, experiment with the keyboard a bit. You can type whatever you like and it won't hurt the computer at all. If the computer stops producing letters in response to your typing, just press the 'reset' button at the back. The joystick is really good to mess about with—and you'll see how handy it is later on. In the meantime let yourself get to know the computer. It's got a lot to offer you.

Note: When you are just using the computer for word processing (see pages 38-42 and the tables on pages 36 and 37), you don't need to insert the IS-BASIC cartridge. In order to write BASIC programs, however, you need to have this cartridge plugged into the ROM BAY on the left-hand side of the machine.

Try typing in the contents of the box below. Computers are a bit funny about little mistakes, so check your typing before you finish. *Remember that you must press the key marked 'enter' at the end of each line.* Don't forget the numbers which begin the lines, they're important too. However, you needn't worry about the blank spaces which appear after the numbers. The Enterprise can put spaces in automatically, to make programs look neater. Notice that computers use a special symbol for nought (Ø), to distinguish it from the capital letter O.

```
1ØØ     GRAPHICS
11Ø     PLOT 64Ø, 36Ø,
12Ø     !
13Ø     DO
14Ø         FOR RADIUS=25Ø TO 1 STEP-16
15Ø             SET INK RND (3)+1
16Ø             PLOT ELLIPSE RADIUS, RADIUS,
17Ø             PLOT PAINT
18Ø         NEXT RADIUS
19Ø         PING
2ØØ     LOOP
21Ø     !
22Ø     END
```

CORRECTING MISTAKES

If you make a typing error, it is easy enough to correct it. If you're still on the same line as the mistake, pressing the key marked 'erase' will move the red flashing 'cursor' to the left, removing letters etc. as it goes. If you have to go back to a previous line, use the joystick to place the cursor at the end of that line, then erase back as far as your error. You can now type the correction, finish off the line by pressing 'enter', and move the cursor (using the joystick) to the bottom of the screen again, or wherever else you want it. Remember that whenever you press a key, the place where your letter or number will appear is the place where the cursor is located at that particular moment.

Later, in the chapter on 'Editing Programs' (page 33), we shall discuss some more versatile ways of making changes.

MAKING THE PROGRAM WORK

When you've typed in the program, type the word RUN and press 'enter' again. Alternatively, you can press the key marked 'function 1', above the number keys. If you use this key, you will not need to press 'enter', and it's quicker than typing RUN. (For more about 'function' keys, see page 43.) Then watch for a while. If you've made a mistake, the computer will print 'Not understood' (or a similar message) on the TV screen. Don't worry at all if that happens. Press *function key 5 and then function 2*, to take another look at the program—then correct the problem, and try again.

TYPING COMMANDS

Commands which this manual suggests you type into the computer will always appear on the page in capital letters. This is mainly for emphasis, but partly because the computer itself often displays BASIC commands in capitals. However, you can *type* words like RUN (and other BASIC commands) in small letters as well; the computer will understand you perfectly. You don't need to go to the trouble of pressing the 'shift' key to get from small letters into capitals all the time.

Any words you don't understand can be found in the Glossary, pages 209–221.

To stop the program, press 'stop'. You will see the response

```
STOP AT LINE ---          (--- is a number)
ok
_
```

WHAT'S A PROGRAM?

A program is a set of instructions which tells the computer what to do. There's nothing special or magic about that, but a computer can't do anything at all without a program. Programs are very exact and very detailed, but then so is embroidery. And, just like any other skilled pastime, programming can be done seriously, or just for fun.

If you want to, you can re-start the program by typing in CONTINUE and pressing 'enter'. Even better, press the key marked 'shift' and at the same time press the function key numbered 1 (you used this to RUN the program). The program will then resume from the point at which you stopped it.

On the other hand, if you're bored with this you can add to it. Type in the line below.

```
135    SET PALETTE RND (256),
       RND (256), RND (256), RND (256)
```

CLEARING THE SCREEN

Having run this program and stopped it, you will now be left with a picture filling the screen. This doesn't make it very easy for you to read what you're typing in. To tidy up the screen, type TEXT, press 'enter' and the screen will be returned to the full text display. Again, if you want to save time (and effort), press function key 5.

LIST

This time, when you have finished typing in the new line, type LIST. *Function key 2 will also do this.*
By now you should be able to remember to press 'enter' whenever you have finished typing in an instruction or a program line. 'Enter' is the key which tells the computer to do what you have typed. Usually, until you press it, nothing will happen; however, you don't normally need to press 'enter' with the function keys—as you have probably found out.

LIST is a word which will show you the whole of a program on the screen (or as much of it as you can fit on the TV screen at one time). You can see now that your new line has fitted in with the old ones. The whole program is now in numerical order. This is the order in which the computer will carry out your instructions when the program is RUN.

GETTING RID OF A PROGRAM

When you get fed up with a program, simply type NEW, press 'enter', and the program will be gone.

Using this computer you can produce all kinds of sounds and many colours very easily. But it is not just there to make noises and display rainbows. Your computer can draw very fine pictures, make decisions, do big calculations very rapidly, sort things into any order you like and repeat things as often as you want.

PRACTICE PROGRAMS

The programs on the following pages are simple examples of the Enterprise's talents.

Try them all! They are just a few things you can do with this machine. In this book you will find out how to do all of them for yourself—plus a lot more. Don't forget to use TEXT to get a full screen to type onto when you want it. You can also use CLEAR SCREEN to empty the TV screen when it gets full. These commands don't actually remove any program lines from the computer's memory; you can view the whole program again any time you like, by typing LIST.

```
  10    PROGRAM "Fire-tunnel"
  20    !
  30    !      This program draws a
  40    !      multi-coloured tunnel with
  50    !      exploding fireballs.
  60    !
 100    GRAPHICS HIRES 256
 110    LET X = 640: LET Y = 360
 120    FOR R=1 TO 255
 130          SET INK R
 140          LET A=X-R-220: LET A1=Y-R-50
 150          LET C=X+R+220: LET C1=Y+R+50
 160          PLOT A, A1; A, C1; C, C1; C, A1; A, A1
 170          PRINT R
 180    NEXT
 190    FOR BALL=1 TO 100
 200          CALL FIREBALL (256, X, Y)
 210    NEXT
 220    !
 230    END
 240    !
 250    !
1000    DEF FIREBALL (COLOURS, A, B)
1010          SET LINE MODE 3
1020          SET INK RND (COLOURS)
1030          FOR GO=1 TO 2
1040                FOR AROUND=1 TO 650 STEP 30
1050                  PLOT A, B, ELLIPSE AROUND,
                    AROUND
1060                NEXT
1070          NEXT
1080          SET LINE MODE 0
1090    END DEF
```

```
100    !       This program will draw boxes.
110    !
120    !       100-140 are comment lines.
130    !       You don't have to type them.
140    !
150    CLEAR SCREEN
160    PRINT AT 5,11: "THIS PROGRAM WILL"
170    PRINT AT 6,10: "DRAW BOXES FOR YOU."
180    PRINT AT 8,1: "The program will ask you to
       type in"
190    PRINT AT 9,1: "some numbers, in pairs. The
       first"
200    PRINT AT 10,1: "number of each pair should
       not be more"
210    PRINT AT 11,1: "than 1279, and the second
       should not"
220    PRINT AT 12,1: "be more than 719. Press
       'enter' "
230    PRINT AT 13,1: "after typing each number."
240    FOR A=1 TO 5000
250    NEXT A
260    !
270    !              Lines 240-250 make the computer
280    !              wait for about 10 seconds.
290    !
300    DO
310        CLEAR SCREEN
320        INPUT AT 5,5, PROMPT "Numbers for
           one corner: ":X
330        INPUT AT 6,29, PROMPT " ":Y
340        INPUT AT 8,5,PROMPT "Numbers for
           opposite corner: ":V
350        INPUT AT 9,34, PROMPT " ":W
360        PRINT AT 11,5: "For how long should the
           box"
370        PRINT AT 12,5: "be displayed?"
380        INPUT AT 14,5, PROMPT "Seconds: ":
           TIME
390        !
400        !      Lines 450-480 are the
410        !      instructions for drawing the
420        !      box and holding it on the
430        !      screen for the time you want.
440        !
450        GRAPHICS
```

```
460           PLOT X,Y;X,W;V,W;V,Y;X,Y
470           FOR B=1 TO 500*TIME
480           NEXT B
490           TEXT
500           PRINT AT 15,18: "More?"
510           DO
520             INPUT AT 17,17,PROMPT "y or n:"
                :ANS$
530           LOOP WHILE ANS$<>"y"AND
              ANS$<>"n"
540       LOOP WHILE ANS$="y"
550       !
560       END ! This is the end of the program.
```

```
100     !       This program sorts 10 numbers
106     !       into numerical order.
108     !
110     NUMERIC ARRAY(1 TO 10)
120     NUMERIC VAR,NUM,BIG
130     CLEAR SCREEN
150     PRINT AT 10,10:"NUMBER SORT"
160     FOR N=1 TO 10
170             PRINT AT 14,10:"TYPE NUMBER";N;
180             INPUT PROMPT ": ":ARRAY(N)
190             PRINT AT 14,25:"                    "
200     NEXT N
210     CLEAR SCREEN
220     PRINT AT 20,20:"SORTING..."
240     LET FIN=10
250     FOR X=1 TO 10
255             LET BIG=0
260             FOR Y=1 TO FIN
270                     IF ARRAY(Y)>BIG THEN LET
                        BIG=ARRAY(Y)
280                     IF ARRAY(Y)=BIG THEN LET
                        NUM=Y
290             NEXT Y
300             LET VAR=ARRAY(FIN)
310             LET ARRAY(FIN)=BIG
320             LET ARRAY (NUM)=VAR
340             LET FIN=FIN-1
350     NEXT X
355     CLEAR SCREEN
360     FOR X=1 TO 10
370             PRINT ARRAY(X)
380     NEXT X
390     END
```

```
100    !          This program gives the
110    !          area/circumference of circles.
120    !
130    LET A$=" of the circle is:"
140    LET B$="Type the radius of the circle: "
150    NUMERIC RADIUS,AREA,CIRCUM
160    DO
170         CLEAR SCREEN
180         PRINT AT 10,10:"1) AREA"
190         PRINT AT 11,10:"2) CIRCUMFERENCE"
200         PRINT AT 12,10:"3) QUIT"
210         DO
220              PRINT AT 15,10: "Type the
                 number"
230              INPUT AT 16,10,PROMPT "of your
                 choice: ":NUM
240         LOOP WHILE NUM<1 OR NUM>3 OR
            NUM<>INT(NUM)
250         CLEAR SCREEN
260         IF NUM=1 THEN
270              INPUT AT 10,1,PROMPT
                 B$:RADIUS
280              LET AREA=PI*RADIUS^2
290              PRINT AT 15,5: "The area";A$;
300              PRINT AT 16,4:AREA
310              FOR X=1 TO 5000
320              NEXT X
330         ELSE IF NUM=2 THEN
340              INPUT AT 10,1,PROMPT
                 B$:RADIUS
350              LET CIRCUM=2*PI*RADIUS
360              PRINT AT 15,1: "The
                 circumference";A$
370              PRINT AT 16,1:STR$ (CIRCUM)
380              FOR X=1 TO 5000
390              NEXT X
400         END IF
410    LOOP WHILE NUM<>3
420    END
```

ALTERING PROGRAMS

Try altering the programs if you want to. It's not a good idea to change the spelling of program instructions, because the computer won't understand if you do. But where words appear between inverted commas you may change them without messing up the program itself. Numbers can also be changed. By doing this you may be able to work out what the program is doing—changing a number will often affect the number of times something is done, or the position of characters on the screen. Some numbers within programs are *codes*—that is, they stand for something else; a colour for instance. You will learn all you need to about these further on in the manual (try page 95 for colour and 1Ø4 for other codes if you like).

Remember always that whatever you type will do the computer *no harm at all*. The worst that can happen is that you will type in something the computer fails to understand. For example, in the number-sorting program (page 11), you might have typed:

```
25Ø FOR X=1 TOO 1Ø
```

In that case, after running the first part of the program, the computer would stop and the screen would show:

```
*** Not understood.
25Ø FOR X=1 TOO 1Ø
```

Error messages are explained in detail on pages 204-208.

STOPPING AND STARTING

If you get completely stuck and want to start again, press the 'reset' button at the back of the machine. This will put you back where you were when you switched the machine on, except that the computer will remember any program you have just typed in. This is best kept for emergencies, though. Normally you will use the 'stop' or 'hold' key for stopping a program that is running. When you use 'reset', you will have to type RUN again to make the program restart.

The 'hold' key is simply used to hold up the program and freeze the action at whatever point you have reached. This is handy if you want to look at a moving picture in a graphics display—or even if you just need a tea break in the middle of a difficult game of Space Invaders. To carry on with the program, simply press the same key again. Notice that 'hold' is very useful if you are LISTing a long program which disappears off the screen, and you want to halt the motion so as to examine some particular line.

The 'stop' key is used if a program is running and you want to halt it so as to modify (or erase) it, or to LIST and examine it before typing CONTINUE to make the program resume.

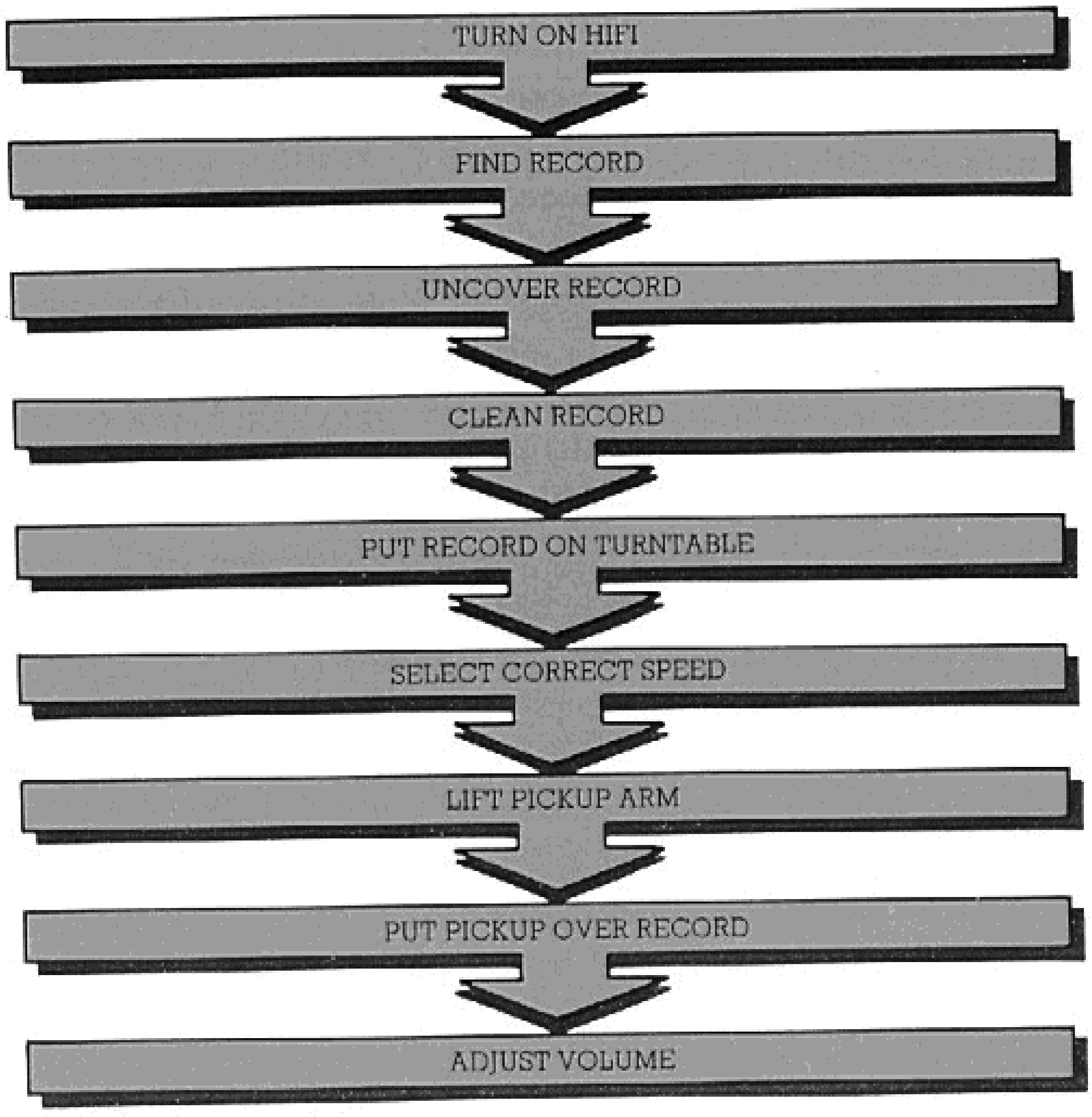
ABOUT PROGRAMMING

You've now had some fun with the machine and, hopefully, introduced yourself. You probably didn't see how some of the programs worked, so this is where we begin adding knowledge to enjoyment.

You've already learned in brief what a program is, but there is a little more to writing it than just giving instructions. You can think of a program as a way to solve a problem using a computer. A computer is either a tool which can expand and speed up your brain power, or it is a pleasurable thing to own on which you can play games or invent some for yourself.

All computers understand instructions, usually in the form of words or lists of numbers. We will be using words to communicate with the Enterprise. Each word is a small instruction; you can put them together like a puzzle or a story to make up bigger instructions and, eventually, complete tasks. That is what programming is essentially about. Giving the computer instructions—in the order in which you want them to happen.

The diagram below shows (using an everyday example) how one task must be broken up into several small ones to make up a program.



COMPUTER LANGUAGES

Now you know what a program is, let's move on a stage further. Just as there are many different ways to talk to another person, so there are many ways to program a computer. And in the same way that there are human languages, there are computer languages.

Languages are made up with definite limits and types of task in mind. Some are especially for programs involving long lists of things, others are particularly good at making pictures with a computer. Others still are there to teach people how to program.

BASIC

The language you are learning through this manual is known as BASIC. It uses words with similar spelling and meaning to English words. It is therefore very easy to learn and understand, even if you are inexperienced with computers. All the programs in this manual are in BASIC, and it is the language the computer understands as long as the IS-BASIC cartridge is plugged in. Remember that, from now on, all the instructions and all the information in this manual relate to BASIC. Some other languages are totally different in both philosophy and approach, and they usually look completely different from BASIC.

Look at this:

```
1Ø PRINT "Hello!"
```

You will know by now that it's a program line—it's one small task which could make up part of a bigger one.

Just as, in a story, each sentence tells you something, so each line in a program tells the computer to do something. If you want to tell the computer to carry out a task, you will need several lines to do it.

DOING THINGS IN ORDER

The number at the beginning of a BASIC program line makes an important difference.

Type in PRINT "Hello!". The display you get when you have pressed 'enter' will tell you what happens. The computer does what it is told straight away—it puts 'Hello!' on the screen, right under the words you have just typed. When you entered programs before, the computer waited until it was told before running the program. It is the line number which makes this difference. Without a line number, the computer carries out your instructions as soon as you press 'enter'. If you add a line number, you must type RUN (or press function key 1) to make the computer work.

So you now know that all BASIC programs are broken up into lines, and that each line needs to have a line number.

If you enter a program line that has the same number as one which you entered earlier, the previous line will automatically be erased from the computer's memory. If you ever want to get rid of a line, simply type in its number followed by 'enter'. The line will disappear, because effectively you have entered an empty line! You can also remove a whole group of lines; the command DELETE 1Ø TO 1ØØ would remove all lines from 1Ø to 1ØØ inclusive.

The order in which lines are put into the computer does not matter. You have already seen how the machine sorts them into numerical order. It's the numerical order which is important.

The line numbers are how the computer decides which instruction to do next. It starts with the lowest number and works its way upwards—*unless told by the program to do otherwise*—until it reaches the end.

The 'unless told by the program to do otherwise' is, as you will see later on, very important. There are several ways in which you can put a program together. Some of these mean that the computer will not follow line numbers in order, because some BASIC words tell it to use other lines instead.

This means that you have to know exactly which things you want the computer to do in what order—which is no more than being sure you know what you want to do before you begin to write a program.

IMMEDIATE MODE

If you type in commands without giving them a line number, it is called *immediate mode*. You can use the computer as a calculator in this way. Here's how.

On the keyboard you will notice these symbols: +, *, =, − and /. Of these, * and / may be unfamiliar to you. They stand for 'multiply' (* instead of ×) and 'divide' (/ instead of ÷). If you type

```
PRINT 2+2
```

you will get the answer, 4, as soon as you press 'enter'. With all the mathematical operators, as they are known, you can do sums on the computer in this way. You can also work out square roots. Try

```
PRINT SQR(1ØØ)
```

and the answer will be the square root of 1ØØ. The word SQR is a special one which the computer understands to mean 'the square root of the number in brackets'. There are several words like this in BASIC. You can find a list of them in the Reference Section, under the heading 'Built-in Functions and Variables'. More of them will be explained as they're used in the text.

Immediate mode will come in useful when you need to work out the results of calculations for use in programs. Using immediate mode does not affect any program lines you may already have typed, and you do not have to do anything special to begin using it. Just miss off the line number.

Many BASIC commands will work under immediate mode as well as within a program. You will see what this means by experimenting. Look through the manual and find BASIC words. Then, using the examples included in each part, try the words in immediate mode. The keyword reference list (page 151) will also tell you which commands can be used in immediate mode.

KEYWORDS

Now you know about programming in BASIC and about immediate mode, let's explain more about keywords. PRINT is one of these, and has a special meaning in BASIC. Keywords are the instructions to which the manual has referred before. There are many of them, and each one tells the computer to do one (or possibly more than one) small thing. PRINT, for example, tells the computer to put a display on the screen.

To make the computer do complicated jobs, you use many keywords and other pieces of information put together as a program. Perhaps you can think of it as telling the computer a story or teaching a child his alphabet—it has to be detailed, exact and in the proper order.

STRINGS

As you have already seen, it is possible to make the computer print a message (Hello!) on the screen by typing the message within inverted commas (" "). This is regarded as a *string* by the computer (see page 52 for more detailed information about strings). This may seem very confusing now, so here are two points which may help you to understand it.

First, 'string' is just a word for a particular type of information which the computer will handle. Secondly, because strings appear in a program in inverted commas, you can think of them as being like quoted speech. In the PRINT "Hello!" statement, the computer literally quoted what you had put between inverted commas.

CHARACTERS

The word 'characters' has appeared from time to time. A character is a letter, number or any other shape provided by the computer for display/communication purposes. All the characters the computer provides are together called the *character set*.

VARIABLES

Now that you know all about program lines, immediate mode and keywords, let's bring something else into the picture.

Imagine two boxes, labelled X and Y. Into each box we can put a number. Later, this number may be taken out and replaced with a new one.

We are now going to tell the computer that each time we RUN the following program and put a pair of numbers into the two boxes, we want it to add together the contents of X and those of Y.

```
1Ø    !
2Ø    !           6Ø and 7Ø ask you to type in
25    !           numbers. They wait for your
3Ø    !           answer. Press 'enter' when you
4Ø    !           have typed each number.
5Ø    !
6Ø    INPUT PROMPT "Type in number X: ":X
7Ø    INPUT PROMPT "Type in number Y: ":Y
8Ø    !
9Ø    !           Line 12Ø adds up numbers X and
95    !           Y and puts the sum
1ØØ   !           into a variable called Z.
11Ø   !
12Ø   LET Z=X+Y
13Ø   !
14Ø   !           Line 16Ø displays the answer.
15Ø   !
16Ø   PRINT X; "+"; Y; "="; Z
17Ø   END
18Ø   !
19Ø   !           Line 17Ø tells you and the
2ØØ   !           computer where the end of the
21Ø   !           program is.
22Ø   !
```

You may not even need to RUN this to see what it is doing. It's something you might have come across at school (or, if you're a parent, in your children's maths books). Do you see how you have to tell the computer *everything* it must do?

COMMENTS IN PROGRAMS

The program lines which begin with exclamation marks are there for your benefit—the computer remembers them but doesn't act on them. They are comments which can be used to tell you what each part of a program is doing. An exclamation mark may also be used to set aside the rest of a program line for comments after one or two instructions on the same line. Alternatively, instead of an exclamation mark, the word REM (for 'remark') may be used—but this must always come immediately after the line number. The comments are best set out as in the example above, so that you can pick them out quickly.

Note how the use of comments makes the program easier to understand. They are not essential, but they help to make a program understood by humans as well as computers.

INPUT PROMPT

INPUT PROMPT (see the program on page 20) is the set of keywords which tells the computer to ask you a question and then wait for you to enter an answer.

If you use INPUT by itself, without adding PROMPT followed by a few words in inverted commas (which are there for your benefit), the computer will ask its question merely by displaying a question mark and the red cursor. The words after PROMPT can be anything you like; and are just there to remind you of what you are supposed to type in. They must always be entered between inverted commas (although these don't appear on the screen). Another way of giving an 'input prompt' is this:

```
10 PRINT "Type in number X ";
20 INPUT X
30 PRINT "Type in number Y ";
40 INPUT Y
```

This method, as you can see, is a little more long-winded than INPUT PROMPT, but the result is still the same. That short program does exactly the same thing as lines 60 and 70 of the program above, except that the program here will allow the computer to print its own little 'input prompt', which is a question mark. An INPUT PROMPT statement stops the computer from printing its question mark.

VARIABLES AGAIN

Variables are names for numbers whose values (i.e. their sizes) may change. For X and Y you might type in any number you like—be it 0.00004 or 400000. If X stood for the number 400000, we would say that four hundred thousand was *the value* of X.

Often you will need to use variables because you will not know in advance what a number will be.

Look back on the first pages of the manual. All the programs you tried out then had variables in them, sometimes because the computer was making its own numbers out of other numbers and sometimes because you decided them while the program was running, by typing them in.

At other times it is convenient to use variable names for very long numbers which will be used several times. This means you don't have to keep on typing them again each time you want them to be used in a program.

Here's another use for variables.

```
10 LET A=Ø !              A begins as Ø.
20 DO !                   DO/LOOP begins.
30      LET A=A+1 !       Add 1 to A.
40      PRINT A, !        Display A's value.
50 LOOP UNTIL A=2Ø !      Go back, repeat loop.
60 !                      Loop ends when A=2Ø.
70 END
```

It's a counter, using something called a DO/LOOP. Each time the Enterprise goes round the loop, it increases the number in box A, by 1. Then, at the end of the loop, it checks to make sure that the value of A is not 2Ø. When A is 2Ø, the program ends. The comma next to 'A' in line 4Ø makes the screen look tidier— you can find out what it does by missing it out and running the program a second time.

DO/LOOPs are not the only way of asking the computer to repeat itself. Page 59 and the Reference Section will tell you about some more.

I NAME THIS VARIABLE...

X, Y and Z are just three of the many names you can give a variable. They don't have to be just one letter— you can use more than thirty if you want.

So you can give all your variables appropriate names –NAME$ for someone's name (this is a *string* variable and is explained on page 53), SUM for the result of adding two numbers.

Note that it makes no difference whether you call a variable PRICE, Price, price or pRicE. The computer doesn't distinguish between capital and small letters in variable names or keywords.

When you write programs of your own you will quickly see how the variable names can help you to read through the program later. This is very important if you need to find errors, make changes or take out part of a program for use in another program—all of which you will want to do eventually.

Incidentally, you could use two variables with very similar names—e.g.:

```
VAT_SUBTOTAL_JUNE_ACCOUNT_NO1
and
VAT_SUBTOTAL_JUNE_ACCOUNT_NO2
```

and the Enterprise would still be able to tell the difference at a glance. The problem is—would you? It's not very clever to use huge variable names like these all the time. Occasionally, you may feel it's appropriate. But eight or ten characters are usually enough to help you tell the difference and know quickly which variables stand for what numbers.

So variables are a way of calculating with numbers whose actual value is unknown to you. The name you give to each one tells you what that number is related to. The computer will tell the difference between variable names of up to 31 characters.

A 'character' here means: a letter or a number (or digit to be absolutely correct). The computer will not understand you if you put spaces in variable names, nor will it understand operators (+, = etc.) or punctuation marks (with the exception of the full stop and the underline marker, '_', which is a good substitute for a space). You must also always begin variable names with a letter, *not* a number or punctuation mark. Here are some 'legal' variables—i.e. those the computer will accept:

```
number         A2$          TOTAL$        Hello1
SUB_TOTAL_3               Name          A1234
```

Notice that you can use a dollar sign. This has a special meaning as it sets aside that variable for use as a string variable. Strings are explained briefly on page 19 and are explained in more detail on pages 52-58.

Here are some the computer does *not* understand:

```
my variable             2A               !!!A$
< >3NOW                  3*THIS NUMBER
```

DECLARING VARIABLES

In the counter program, the first line read LET A=Ø. Try the program again without it. It doesn't work, does it?

However, if you are using an INPUT statement to tell the computer to let you type in the value of a variable (as in the program which added two numbers of your choice), you don't need to declare this variable by using LET.

USING VARIABLES

Try typing LET A=3 instead of LET A=Ø at the beginning of that program. It will tell you a little more about how variables work.

The computer expects the variable which is changing or being declared to follow immediately after LET. So if variable S=Ø and variable G=1Ø, you could type LET S=G to make S into 1Ø. G would still contain 1Ø but the computer would assume it was as big as variable S.

LET introduces the computer to a variable—you're saying 'computer, here's a variable called FIRSTNUM; at the moment, it equals Ø' or whatever number you want that variable to be. It is not always necessary to use LET, but it is generally better if you do. It makes your variables easier to spot right from the start.

If you want to, you can use a BASIC word as a variable name. But if you do, you *must* use LET to tell the computer to look out for a variable with the same name as one of the words it understands. There are some BASIC words which cannot be used as variable names—a bit of experimentation will soon tell you which ones these are. (A more versatile way of declaring variables is explained on page 73, in the chapter on storing information.)

Overpage is another program illustrating all the principles you've just read about. This time each line is explained in the light of your new knowledge about these variables.

```
1Ø    LET A$=" The sum of the two numbers
      is: "
2Ø    !
3Ø    !            1Ø declares string variable A$.
4Ø    !
5Ø    LET SUM=Ø
6Ø    !
7Ø    !            SUM is the variable which will
75    !            contain the result of the
8Ø    !            addition below. It begins as Ø.
9Ø    !
1ØØ   INPUT PROMPT "Please type in your
      first number: ": FIRSTNUM
11Ø   INPUT PROMPT "Type in your second
      number: ": SECNUM
12Ø   !
13Ø   !            1ØØ and 11Ø ask you to type in
135   !            numbers to be added. You do not
14Ø   !            need to use LET, because the two
15Ø   !            new variables are being 'input'.
16Ø   !
17Ø   LET SUM=FIRSTNUM+SECNUM
18Ø   !
19Ø   !            SUM (which was Ø) now becomes
195   !            the result of the addition of
2ØØ   !            FIRSTNUM and SECNUM.
21Ø   !
22Ø   PRINT A$;SUM
23Ø   !
24Ø   !            Line 22Ø tells the computer to
245   !            display the sentence from line
25Ø   !            1Ø and the value of SUM all on
26Ø   !            the same screen line.
29Ø   !
3ØØ   END
```

This is an *expression*: 4+2*3−5

These are *operators*: ^, +, −, *, /, =, <, >, <>, <= and >=

All of the above operators can be used on the Enterprise. Some of them will be obvious to you, but others may not. The symbols +, −, *, / and = should already be clear to you; the computer uses * instead of × and / instead of ÷.

Here are the rest, just to make sure:

^ means 'to the power of' or 'involution'. 2^3 is 2 cubed.
< means 'less than', e.g. 2<3
> means 'greater than', e.g. 3>2
<> means 'greater or less than', or 'not equal to'.
<= means 'less than or equal to'.
>= means 'greater than or equal to'.

The last five operators are known as 'relational operators'. Their main use on a computer is in handling variables—for instance:

```
10    INPUT PROMPT "Please type in a
      number: ": A
20    !
30    !         Line 10 asks for a number which is
35    !         to be evaluated.
40    !
50    IF A<0 THEN PRINT "This number is
      negative."
55    IF A=0 THEN PRINT "This is zero!"
60    !
70    !         50 makes a simple decision. If
75    !         A is a negative number (e.g. −9),
80    !         say so. IF is the crucial point.
90    !         If number A does not fit
100   !         that line, the computer looks
110   !         for a line which A does fit.
120   !         A<0 in line 50 literally
130   !         means 'A is less than 0'.
140   !
150   IF A>0 AND A<50 THEN PRINT "This
      number is more than 0 and less than 50."
160   !
165   !         150 is another 'IF/THEN'. It's
```

```
170    !            like English—IF you don't want
180    !            to come THEN go home. Now the
190    !            computer looks to see if A is more
195    !            than Ø but less than 5Ø. If not, it
200    !            looks at the line below.
210    !            ________________________________
220    IF A > 5Ø AND A < 1ØØ THEN PRINT "This
       number is more than 5Ø and less than 1ØØ."
230    !            ________________________________
240    !            Line 3ØØ is the last decision line.
245    !            It looks to see if A is more
250    !            than 1ØØ.
290    !            ________________________________
300    IF A > 1ØØ THEN PRINT "This number is
       bigger than 1ØØ."
310    END
```

The program will decide what category your number comes into, according to the instructions it receives. That is one of the ways in which the computer makes decisions. There are several other ways in which this can be done. They're explained on page 64.

Relational operators are a way of comparing numbers. They do not change numbers at all. The operators referred to below all change numbers in some way.

OPERATOR PRIORITY

Look again at the expression: 4+2*3−5

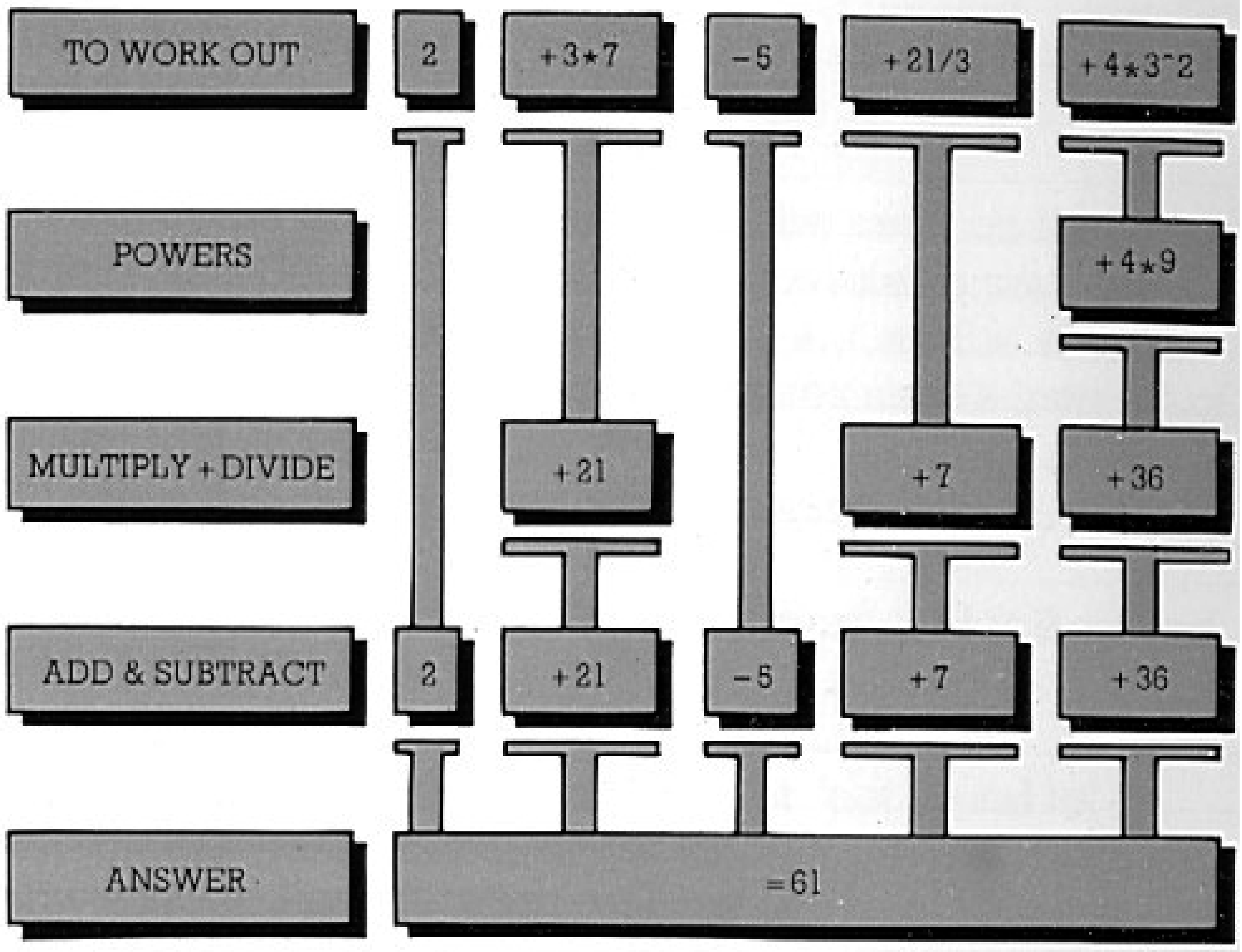
What do you think the result should be? Thirteen, because you work out each part from left to right? Well, it's five, and here's why.

Powers are solved first. If the expression had had 4^3 in it, this would have been calculated first. Then come multiplication and division. 2*3 or 6/2 would be calculated before any addition or subtraction was done by the computer.

Multiplication and division have the same priority, and if, for instance, an expression contained two divisions and one multiplication, the machine would work them out in order from left to right.

This done, the computer drops down to the next level of priority and starts on the addition and subtraction. These are also worked out from left to right and, by this time, you should have a result—in this case 5.

Let's look at a longer one now, and take it to bits in the way the computer would.



2+3*7-5+21/3+4*3^2 is our expression.
First work out the powers.
3^2=9
Now it looks like this:
2+3*7-5+21/3+4*9
Then do the multiplications and divisions—the second level.
3*7=21   21/3=7 4*9=36
Now you have:
2+21-5+7+36
And this is worked out from left to right, so:
2+21=23   23-5=18   18+7=25   25+36=61
So the result is 61.

This, as you can now see, will affect the way your calculations might work out. You might expect one result where the computer gives another. If you want, you can change the priority and have different sections of an expression given 'express treatment' by the computer. All you do is stick brackets round the part you want evaluated first. Look at the difference this makes to the expression above:

(2+3)*7-5+21/3+4*3^2
The computer treats (2+3) as a unit on its own and so works it out first. Thus:
5*7-5+21/3+4*3^2

Then it works as normal, and does the powers
3^2=9
so:
5*7-5+21/3+4*9

Then multiplication and division—from left to right:
5*7=35 21/3=7 4*9=36
leaving
35-5+7+36
to figure out:
35-5=30 30+7=37 37+36=73

The result is now 73 and not 61 as before. So, as you can see, numbers can be juggled about in all sorts of ways using a computer. Don't forget that, inside each pair of brackets, the computer will deal with expressions by using the same priority system. So if you have (2+3*4) the multiplication will come before the addition.

Try any of the examples above. The simplest way is to use PRINT in immediate mode: e.g. type PRINT 5*7-5+21/3+4*9.

Lastly, before we go on, you can use numbers with decimal points (e.g. 0.23345 or .0098) and negative numbers (e.g. -2) as and when you want to.

You may be finding all these numbers a bit stuffy. Once you've got used to using numbers the way a computer does, though, you won't notice. Don't forget that, although you will never need to be a mathematician to program a computer expertly, you will always need to use arithmetic. Even graphics commands need numbers which have to be worked out—though often this is very easy.

A semicolon will tell the computer that the things on either side of it are to be printed next to each other on the screen. A program with four separate PRINT statements but no semicolons would appear like this:

```
Hello,
I'm
your
computer.
```

instead of:

```
Hello, I'm your computer.
```

There are other ways of putting things onto the screen to make them look nice.

Try using a comma, as below (this modifies the counter program used on page 22):

```
 10  LET A=0 !                      Declare A as 0.
 20  DO !                           Begin DO/LOOP.
 30  LET A=A+1 !                    Add 1 to A.
 40        PRINT A,
 50        !
 60        !     Print A. The comma tells the
 65        !     computer to put A eight character
 70        !     positions along from the last A.
 80        !
 90  LOOP UNTIL A=20 ! Loop again if A < 20.
100  END
```

See how the comma arranges things into columns? The screen is normally divided by the computer into 40 'character positions' widthways and 24 from top to bottom. The comma will normally separate items by eight character positions.

Using a comma to format text is a little like using the semicolon, but it adds space between one string or number and the next. The semicolon just prints things next to each other across a line.

Try this short program:

```
10  PRINT "There was an old man from St. Bees"
20  PRINT
30  PRINT "Who was stung on the head by
    a wasp."
```

```
 40  PRINT
 50  PRINT "When asked, 'Does it hurt?' "
 60  PRINT
 70  PRINT "He replied," "No, it doesn't "
 80  PRINT
 90  PRINT " 'It can do it again if it likes!' "
100  END
```

Lines 2Ø, 4Ø, 6Ø and 8Ø all print an empty line on the screen—like a carriage return on a typewriter. The word PRINT on its own on a line means 'print an empty line' or 'jump down a line and use the next one instead'.

It is not possible to print double inverted commas on the screen in the same way as other characters (e.g. PRINT " " "), because this confuses the computer. Instead, type the double inverted commas twice in order to get it printed once. Try line 70 above as follows:

```
7Ø PRINT "He replied," "No, it doesn't"
```

PRINT AT

The next way to set out text is with PRINT AT. You already know what PRINT does. You also know the meaning of the word AT—it tells the computer where on the screen to put some characters.

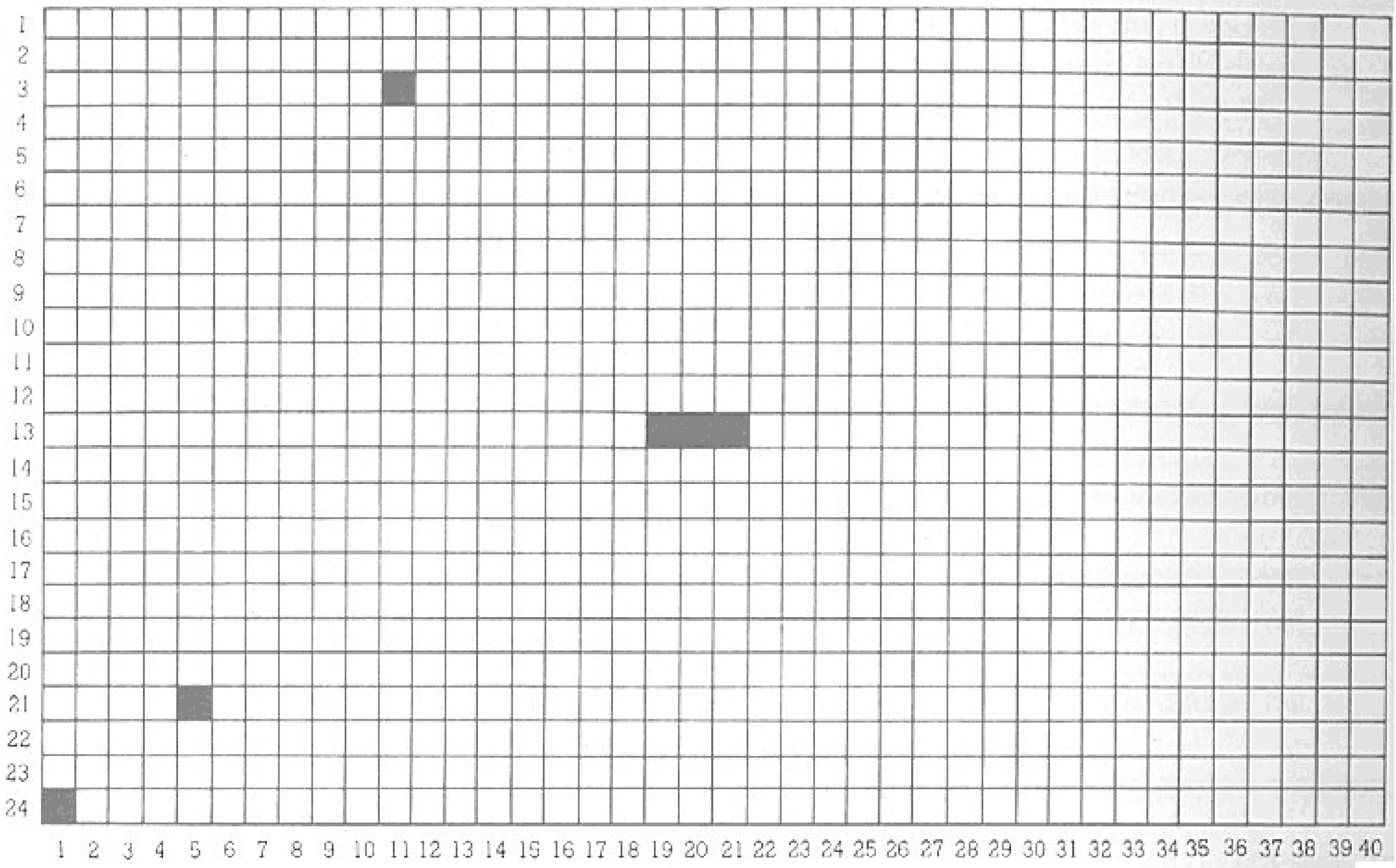
Try this:

```
PRINT AT 2Ø, 2Ø: "I'M OVER HERE"
```

Imagine the screen is divided up into positions; 4Ø across and 24 down. Any of these positions can be referred to with two numbers. The string above is printed 2Ø lines down, and 2Ø positions (or columns) across.

PRINT AT 1Ø, 2Ø repeats the string higher on the screen.

The diagram opposite shows you how the screen is normally made up of 96Ø (24*4Ø) character positions; it should help you to work out *where* any position is on the screen. You select your PRINT AT position by telling the computer how far down the screen (row number) and how far across (column number) you want to place your string. A character position is an 'imaginary' square on the screen, into which a single



character will fit. A comparable but differing system is used with graphics. See pages 90-91.

Some more advanced ways to format text are explained on page 89.

8Ø-COLUMN SCREEN

The command TEXT 8Ø enables you to type up to 8Ø characters on each line, instead of only 4Ø; the characters will now be narrower, of course, and may not be clear on your TV screen. (The command simultaneously clears the screen.)

TEXT 4Ø returns you to 4Ø-column typing.

EDITING PROGRAMS

You already know how to correct typing errors by using the joystick and the 'erase' key. We shall now look at some more elaborate ways of altering programs—inserting new lines, changing the line numbers, altering parts of lines, and so on. Sometimes this is quite a complicated matter. To make it easier, the computer provides 'word processing' facilities, which will be partly introduced in this chapter and discussed more fully in the next.

Let's go through the program-editing commands one by one.

RENUMBER

When you typed in programs earlier in the manual, you may have noticed that the line numbers often began at 1ØØ and went on: 11Ø, 12Ø, 13Ø... We could have used 1, 2, 3, 4... instead.

The reason why you don't normally use 1, 2, 3, 4... for line numbers is simply that you may want to put in more lines later—just as if you were writing a story and you got halfway through before realizing you'd left something out at the beginning. You can't have line number 2.5, but you can (if you get stuck this way) type RENUMBER, and the computer will change all the line numbers. If you type RENUMBER STEP 1ØØ, the lines will increase by 1ØØ at a time. Just typing RENUMBER, with no number following it, will make the computer increase the line numbers by 1Ø (starting from 1ØØ). This version of the command is also obtained by pressing 'shift' with function key 3.

STEP

STEP is a keyword which appears from time to time in BASIC, in conjunction with some other keyword (such as RENUMBER in the case above). STEP means 'in steps of...'. STEP 1ØØ means in steps of 1ØØ—i.e. 1ØØ, 2ØØ, 3ØØ...

AUTO

Another handy little word will make the computer put in the program lines for you automatically. This word is AUTO. Type it in, and the number 1ØØ will appear, on a new line.

You can then type in your program line and press 'enter'. Then 11Ø will appear, and so on. To stop this automatic numbering, just press the 'stop' key.

You can specify the point in a program where you want your auto numbering to begin. AUTO AT 2ØØ will begin auto numbering from line 2ØØ. AUTO AT 2ØØ

STEP 1ØØ will begin from 2ØØ and increase each line after that by 1ØØ. You can use any number from 1 to 9999, which is the largest line number allowed. (Typing AUTO AT 9999 STEP 1Ø, or something similar, will confuse the machine.) Don't forget that if (say) line 25Ø is entered by auto numbering, it will replace any line numbered 25Ø which you had typed in before.

DELETE AND LIST

You have already used the word NEW—it removes the program you are using from the computer's memory. The word DELETE (followed by 'enter') will also do this.

Typing DELETE 1ØØ will remove line 1ØØ from the program. DELETE 1ØØ TO 14Ø will remove lines 1ØØ to 14Ø inclusive.

If you only want to delete one line, it is quicker to type in its number alone and press 'enter'.

In conjunction with DELETE you can use the words FIRST and LAST. Thus, DELETE FIRST TO LAST would erase an entire program—of course, NEW (or DELETE just by itself) is better! But DELETE FIRST TO 1ØØ is quite sensible if you want to delete the lines up to and including 1ØØ of a big program.

You can also delete smaller pieces of a program, i.e. particular words or characters within a program line. Table 1 at the end of the chapter explains how this is done. To find a line so as to alter it, type LIST. This word by itself (as you know) will display the whole program. LIST followed by a particular line number will make that line appear on its own just above the cursor. Typing LIST... TO... will display a group of lines from the first to the second number, inclusive. FIRST and LAST can be used with LIST in the same way as they can with DELETE.

Then you need to position the cursor at the beginning or end of the portion of the line that you want to remove. You do this, of course, by using the joystick. Once the changes to the line have been made, press 'enter' while the cursor is still on this same line. This will re-enter the modified line in place of the original one. (Note that until you have done this, the computer still 'remembers' the line in its original form.)

OVERWRITE OR INSERT

If you want to alter something in the middle of a program line, you don't necessarily have to use the 'erase' or 'del' key. As long as the machine is working in what is known as *overwrite mode*, any character that is under the cursor is automatically deleted when you type in a new one—you 'write over', and replace, the old material. So to change the number 234 into 567, simply place the cursor over the 2 and type in the three new characters.

If you want to put in new characters without at the same time deleting any old ones, you can put the computer into *insert mode*. To switch from 'overwrite' to 'insert' mode or vice versa, hold down the 'ctrl' key and press 'ins'. To show when you are in 'insert mode', the cursor changes its appearance (it takes the form of an arrow pointing left).

In *insert mode*, anything that you type in the middle of a line is fitted in between the characters that were there before—the ones on the right are moved across to make room for it. This will sometimes mean that words will disappear off the right-hand edge of the display. The computer has not forgotten about these words—you can make them reappear by holding down 'ctrl' and pressing function key 1, as described in the next chapter. As long as there are characters 'off the edge' of the screen, the symbol '>' will be seen at the end of the line.

So another way of changing 234 into 567 is to delete the old number and then select 'insert mode' in order to put in the new one.

INSERTING LINES

To insert a complete new program line, simply find out the place in the sequence where you want it to come, and type it in with an appropriate number. The number will tell the computer where the line has to fit in—as you saw very early in the manual.

MOVING THE CURSOR

You know that holding the joystick in one position will make it move the cursor repeatedly in that direction until you release it. And you will have found that when the cursor reaches the top or bottom of the screen, the text will 'scroll' downwards or upwards, to show lines that were previously undisplayed.

If the cursor is moving up or down, pressing 'shift' will make it move a page (or screenful) at a time. This is useful for moving quickly from one part of a long program to another. (Using 'ctrl' in place of 'shift' will move the cursor by paragraphs—you will see that this is useful for word processing.) Pressing 'ctrl' while you are moving the cursor sideways moves it in units of words instead of letters. Pressing 'shift' and moving the joystick sideways will move the cursor straight to the end of the line in the direction you want.

The following two tables show you all the different ways of deleting and inserting characters. Positioning the cursor is always done with the joystick.

Experiment with these functions on a program—perhaps you've already got one in the computer. They may appear a little confusing at first, but once you know which keys perform what functions, you will find that word processing (see next chapter) is very easy.

TABLE 1—DELETION

FUNCTION	KEY (COMBINATION)	EXCEPTIONS/CONDITIONS
Erase character left: Erase character to the left of the cursor and close up line.	ERASE	Moves cursor 1 space left. If it's already in the leftmost position, the line is joined to the previous one.
Erase line left: Erase from start of line up to cursor. Move right part of the line to the left to close up.	SHIFT + ERASE	Leaves cursor in leftmost character position.
Delete character right: Delete character under cursor. Moves right part of the line to the left to close up.	DEL	Leaves cursor in same position. It's the characters which move! If cursor is at end of the line, moves next line up to join it.
Delete line right: Delete from cursor to end of line.	SHIFT + DEL	Leaves cursor in same position.
Erase word left: Erases leftwards until it has erased the left-most character of a word.	CTRL + ERASE	Starts with character position to the left of the cursor. Only letters or numbers are considered part of the word. Intermediate special characters are erased.
Delete word right: Deletes rightwards until it has deleted the right-most character of a word.	CTRL + DEL	Starts with character position under the cursor, otherwise as above.

TABLE 2—INSERTION

FUNCTION	KEY (COMBINATION)	EXCEPTIONS/CONDITIONS
Insert character: Move characters, from cursor to end of line, one position right. Put space under cursor.	INS	When in insert mode this is equivalent to pressing space bar, except that the cursor doesn't move.
Insert line: Move characters from cursor to end of line down by one line of their own.	SHIFT + INS	If cursor is at the beginning of the line, this creates a blank line.
Toggle insert: Swop between insert mode and overwrite mode.	CTRL + INS	A different cursor is displayed, depending on whether the computer is in insert or overwrite mode.
INSERT MODE: Allows you to move the cursor to any position in program or text and type in characters which are fitted in by moving those after them to the right.		
OVERWRITE MODE: New characters typed replace those already there. No space is made by moving characters to the right. You literally 'type over' characters which have been typed before.		

One very valuable thing you can do with your computer is use it as a sophisticated typewriter. You can type in a passage of text—a letter, essay, or whatever—and then print it on a line printer if you have one. The advantage of the computer over an ordinary typewriter is that any mistakes can be corrected much more simply, and that various special functions are provided to help you format your paragraphs and move them around.

THE WORD PROCESSOR

The Enterprise's built-in word processor can be used whether or not the BASIC cartridge is inserted. If the cartridge is plugged in and you want to switch over from programming to word processing, type the word TYPE and press 'enter'. The display on the screen will then change; you will see special wording across the top and bottom, indicating the uses of the function keys for word processing purposes.

This is the same display that will appear if you switch on the computer without the cartridge inserted.

Note that the command TYPE will erase any BASIC program from the computer's memory. Another way of giving this command is by pressing 'shift' with function key 8.

If you have redefined part or all of the character set (see page 105), these definitions will be retained when you enter the word processor. To revert to the original characters type CLEAR FONT and press 'enter' before leaving BASIC.

Now type a few nonsense letters, pressing 'enter' when you think you've typed enough. You will then simply see the cursor move to the next line down; there will be no 'error' message to say that you've entered something the computer doesn't understand. When used for word processing, the computer merely behaves like a typewriter with extra facilities — it displays what you have typed, but doesn't try to understand it.

The 'enter' key is treated by the word processor as a 'carriage return'—but you only need to press it at the end of a paragraph, not at the end of every line. Remember that the computer uses something called *word wrap*, which automatically keeps the lines within the required length.

Note that all the text between two carriage returns (the 'enter' key) counts as a paragraph. Functions that apply to whole paragraphs (see below) all work on the paragraph which currently contains the cursor.

Alterations to the text on the screen can be made in exactly the same ways that you would alter a program—see the two tables at the end of the last chapter. The special functions that we are now going to examine are designed mainly for use when typing a document, though they will also work if you are entering lines of BASIC.

SPECIAL FUNCTIONS

To use any of these functions, press the appropriate function key while holding down the 'ctrl' or 'alt' key.

'CTRL' + FUNCTION 1—RE-FORM

This function will adjust the line lengths of the paragraph that contains the cursor. This is useful if you have inserted or removed several words, and the lines are uneven.

'ALT' + FUNCTION 1—JUSTIFY AND RE-FORM

'Justification' means straightening out the edges of a paragraph so that the beginnings and ends of the lines form a neat vertical column. This is like the text in most newspapers.

'CTRL' + FUNCTION 2—CENTRE

This puts a line or paragraph in the horizontal middle of the 'page'. Useful for titles and headings.

'ALT' + FUNCTION 2—CLEAR ALL TABS

Just like a typewriter, the computer allows you to set tab stops and then press the 'tab' key to skip across the line to the pre-set point. Before re-setting a number of tabs, you will probably want to remove all the previous ones with a single key-press.

'CTRL' + FUNCTION 3—TAB SET/CLEAR

Tab is short for 'tabulation'. This means dividing the screen up into several 'columns' across, for formatting purposes. If you wanted to type in a table of information (like the function key table on page 45, for instance), you could set as many tabs across the screen as there were columns in your table. The 'tab' key can then be used to jump straight from one to the next. This ensures both speed and accuracy in levelling the columns.

Setting a tab is done simply by moving the cursor (using the joystick) to the place where you want the tab to be, and then pressing 'ctrl' with function key 3. If there is already a tab in the cursor position, this same command will remove it.

'ALT' + FUNCTION 3—RULER LINE

The 'ruler line' across the top of the screen shows you where the current left and right margins are, and where any tabs have been set (indicated by vertical strokes). This command switches the ruler line on or off.

'CTRL' + FUNCTION 4—LEFT MARGIN

Of course, sometimes you will want to change the positions of your margins. To set the left margin (the position where the lines will begin), move the cursor across the screen to where you want the margin to be, then press this function key with 'ctrl'.

'ALT' + FUNCTION 4—RIGHT MARGIN

As above, but for the right margin.

'CTRL' + FUNCTION 5—RELEASE MARGINS

This command allows you to insert words or characters outside the current margin settings. The symbol (*) on the status line shows that the command has been given; repeat the same key-press to cancel it.

'ALT + FUNCTION 5—RESET MARGINS

This sets the margins to the furthermost positions on the left and right of the 'page'. It also re-sets the tabs. If you've been typing a narrow column of text in the middle of the screen and then want to move the margins outwards for your next paragraph, press 'alt' with function key 5, then 'enter'. You can then set the margins wherever you want them.

'CTRL' + FUNCTION 6—MOVE UP

With this command, the paragraph containing the cursor is moved up by one line; the line that was above it reappears beneath it. The key-press can be repeated until the paragraph is in the desired position.

'ALT' + FUNCTION 6—MOVE DOWN

This opposite of the above.

'CTRL' + FUNCTION 7—CHANGE LINE COLOUR

This selects a new pair of colours for the text and background on the line where the cursor is placed. On an 8Ø-column screen, there is a choice of four colour-pairs; repeated pressing of the key will show all of them in turn. On a 4Ø-column screen, there are two colour-pairs only.

'ALT' + FUNCTION 7—CHANGE PARAGRAPH COLOUR

As above, but applied to a whole paragraph.

The uses of the function keys on their own (without 'ctrl' or 'alt') vary according to whether you are programming or word processing. Note that in some cases there are 'failsafe' devices—e.g. to change over to 8Ø-column text while using the word processor, you press function key 5 and then asked to confirm your command by pressing 'enter' (or to cancel it with 'esc'). This stops you from erasing a document by pressing the function key by accident.

For complete tables of the function key operations, see page 45.

PRINTING OUT TEXT

Now that you've typed your document, you will probably want to print it on paper (if you have a printer) or save it on cassette for later use.

If you want to use the printer, it should, of course, be connected to the computer through the socket at the back of the machine (labelled 'printer'). It must be switched on, and 'on line'. Now press function key 3. A message on the screen will remind you to make sure the printer is set up correctly. Press 'enter', and your document will be printed.

SAVING TEXT ON CASSETTE

Saving text on cassette is very similar to saving a program—see pages 46-47 for how to connect the computer to the cassette recorder. Like a program, a document that you want to save must be given a name conforming to the rules given on page 46. The only difference is that the document name—the 'filename'—isn't typed between inverted commas.

Press function key 2. The computer will then ask you to enter the filename. After typing (for example) LETTER, press the 'record' and 'play' buttons on the cassette recorder, then press 'enter'. Your text will be saved under remote control if the appropriate connections have been made.

Similarly, the word processor allows you to load a document much as you would load a program (but you will need to take the plug out of the REM socket while rewinding the tape). Simply press function key 1, type LETTER—or whatever name the document has been given—and press 'enter'.

EXITING FROM THE WORD PROCESSOR

When you've finished using the word processor and want to return to BASIC programming, press function key 8. The computer will then ask you what program you want to switch to; type BASIC, and press 'enter'. Of course, if a cartridge other than IS-BASIC is plugged in, you will have to type the appropriate name (e.g. LISP) instead, in order to use it. Typing WP returns you to the word processor while erasing your previous text. If you have pressed function key 8 by mistake, pressing 'esc' puts you back where you were before.

You have already tried pressing some or all of these keys. They are located above the number keys and are themselves numbered 1–8.

These keys can be redefined by you to perform whatever function you may wish them to do. Until you redefine them, however, the computer provides functions for each key which you will probably find useful at some stage.

Redefining the function keys is done by typing (for example):

```
SET FKEY 1 "PRINT"
```

The function you wish the key to perform should come in inverted commas. The number after FKEY is the number of the key being redefined.

Try typing in the command above and then press the first function key. The word PRINT will appear on the screen. This means the keys can be used to put frequently-used keywords on the screen just by pressing one key instead of typing the whole word. Words like RUN, LIST and RENUMBER are possibly more useful because you do not always need to add anything after them. Pressing a function key to list a program is far quicker than typing the word LIST and pressing 'enter'.

A carriage 'return'—the equivalent of pressing 'enter'—can be added to a function key definition, by appending

```
&CHR$(13)
```

to the end of that definition, e.g.:

```
SET FKEY 1 "PRINT" &CHR$(13)
```

The large table on page 45 shows the uses of each function key as they stand when you switch on the computer with the BASIC cartridge inserted. Notice that you can use 'shift', 'ctrl' and 'alt' with the function keys, which means each function key actually carries four different functions. When you want to reset the keys to these functions after redefining them, type CLEAR FKEYS.

When pressed by themselves, as you know, the function keys have different uses when you are operating the word processor. These uses are set out in the smaller table.

FUNCTION KEY OPERATIONS

	BASIC only		BASIC or word processor	
Key	NORMAL	+ SHIFT	+ CTRL	+ ALT
1	START Runs program. If no program, boots (i.e. loads into memory) program from disk, and runs it. If no disk, loads from cassette.	CONTINUE Carries on program after 'stop' command.	RE-FORM Adjusts line lengths to keep them within the margins (tidies up a paragraph after editing).	JUSTIFY AND RE-FORM Evens out spaces in lines, straightens margins.
2	LIST Lists whole program on screen.	LLIST Lists whole program onto printer.	CENTRE Puts text in centre of screen.	CLEAR ALL TABS
3	AUTO Automatic numbering in steps of 10. To 'switch off', press 'stop'.	RENUMBER Renumbers whole program in steps of 10.	TAB SET/CLEAR Sets or removes tab stop at current cursor position.	RULER LINE Line showing margins and tabs is switched on or off.
4	REMOTE 1 Turns on or off ('toggles') Rem 1 control switch.	REMOTE 2 As left but for Rem 2 socket.	LEFT MARGIN Sets left margin to current cursor position.	RIGHT MARGIN Same as for left margin, but sets the right one.
5	TEXT Sets up full screen in text mode, and clears text and graphics pages.	DISPLAY TEXT Switches from graphics to text page, without erasing either.	RELEASE MARGINS Allows text to be entered outside the margin settings.	RESET MARGINS Sets margins to permit typing across full width of screen.
6	GRAPHICS Sets up first 20 lines as graphics, last 4 lines as text; clears both text and graphics pages.	DISPLAY GRAPHICS Switches from text to graphics display, without erasing either.	MOVE UP Positions paragraph above the line that was preceding it.	MOVE DOWN Moves paragraph below the line that was following it.
7	CLICK Turns keyboard click (heard each time you press a key) on or off.	SPEAKER Turns all sound capability off (or on again).	CHANGE LINE COLOUR	CHANGE PARAGRAPH COLOUR
8	INFO Gives information about programs in memory.	TYPE Puts you into word processing mode.		

'NORMAL' FUNCTIONS FOR WORD PROCESSOR ONLY

1. LOAD Permits loading of text from cassette or disk.	2. SAVE Permits saving of text on cassette or disk.	3. PRINT Sends text to printer.	4. HELP Displays information on various word processing functions.
5. TEXT 8Ø Clears text and sets up screen in 8Ø-column mode.	6. TEXT 4Ø Clears text and sets up screen in 4Ø-column mode.	7. CLICK Turns keyboard click on or off.	8. EXIT Exits word processor; permits return to BASIC etc.

HANDLING PROGRAMS ON CASSETTE

In the Setting-Up Guide, you learnt how to control a cassette recorder simply to load a program into the computer's memory. However, you can also use a cassette recorder to store your own programs.

Perhaps you want to save a program which was copied from an earlier part of this section of the manual —possibly you will have made some of your own additions to it, so we'll assume you already have a program in the computer's memory when you begin reading this.

First, make sure the computer and cassette recorder are connected up correctly. The socket at the back of the machine, marked OUT, is the one through which programs are put onto cassette. Plug one of the larger cassette lead plugs into this socket.

Then plug the opposite end of the lead into the socket marked MIC (or something similar) on the tape recorder.

Now plug one of the small plugs into the socket marked REM2, and the opposite end of this plug into the small remote socket on the recorder (probably marked REM)—if your recorder has one.

It does not matter whether the computer is on or off when you do this!

Next, you have to give your program a name. For the purposes of clarity we'll assume you're calling it 'myprog'. The name can be up to twenty-eight characters long, and may contain letters, numbers and the following punctuation marks: '.', '–', '_', ' ' and '/'. You could call the program 'My-program-number-1' if you wanted to.

Type:

```
SAVE "myprog"
```

Press the 'record' and 'play' buttons, then press 'enter'. You should see the message

```
SAVING myprog
```

—and then:

```
OK_
```

when it's finished saving it. Because the remote socket is connected, the Enterprise will automatically stop and start the tape at the right moments, as long as your recorder has a remote control facility.

While the program is being saved on to tape, the sound of this operation will be quietly echoed on the computer's built-in speaker; you will probably find this is a useful indication that the saving is working properly.

MAKING SURE

Of course, you may want to check that your program has been saved onto the tape properly. This is done using the command VERIFY.

First wind the tape back to where it was when you began saving the program. (If the remote control facility is being used, the computer will probably be disabling the rewind on the recorder. If this is so, type TOGGLE REM2 or press the 'shift' key in conjunction with function key 4.) Then type:

```
VERIFY "myprog"
```

and start the tape and press 'enter'. Again, the Enterprise will control the tape recorder.

If the program has been saved without error, the computer will respond with the message:

```
ok
```

will appear. Otherwise, you will see an error message. In this event, you should make sure you have made the right connections, check that the volume control is set at the correct recording level—and go through the saving process again.

MIXING PROGRAMS TOGETHER

If you already have a program in the computer and you want to add another one to it from tape, you can do this with the command MERGE.

MERGE could come in handy if you were in the middle of writing a long program and had saved it as a series of subprograms (or smaller programs which make up part of a bigger one). If you wanted to put the whole program together from all the separate parts you'd saved on tape, you could do this using MERGE.

If you want to fit all the lines from both programs into the computer, you will have to make sure all the line numbers are different. This is why:

Imagine you have one small program in memory.

Its line numbers are 1ØØ, 11Ø, 14Ø and 16Ø.

Now imagine you want to merge in another small program from tape. Its line numbers are 14Ø, 16Ø 18Ø and 2ØØ.

If you tried to MERGE these two programs without RENUMBERing the one in memory, lines 14Ø and 16Ø of the tape program would replace 14Ø and 16Ø in memory. So you would effectively lose two lines from your original program.

So when you merge two programs together, the lines from each program are 'dovetailed' together to make a bigger program.

Here's how to merge in a program from tape.

The method with the tape recorder is exactly the same as for loading (MERGEing is the same as loading, except that loading a program replaces anything else in memory at the time—it 'throws away' the current program and substitutes the new one). *If you are unsure of how to load a program, look at page 11 of the Setting-Up Guide.*

So make sure your tape is at the beginning of the program you wish to MERGE, and that you are sure all the line numbers are right. Then type

```
MERGE "newprog"
```

('newprog' is the program's name—the name under which it was originally SAVEd).

Start the tape and press 'enter'. The programs will then be joined together. Try it!

FILES

Programs stored on tape can also be known as *files*. This also applies to the other program storage medium, disks. A 'file' can also be a collection of information for use by a computer. This is called a *data file*.

DISKS

If disks are attached to the computer, all the tape commands can be used, but the disks will automatically be used instead. All the operations of saving, loading etc. will then take place much faster.

STRINGS

The earlier part of the manual dealt with strings in very little detail. Now that you have learnt the fundamental concepts, a word about strings will put your knowledge into perspective.

Strings are one of the two main types of information—numbers being the other—that the computer can handle. In fact, handling information is what a computer is all about. It has probably become obvious to you by now that a computer does not have the ability to think as a person does. It does exactly what you tell it to, with information *you* give it. So, at root, programming is just a way of sorting, collecting and manipulating information.

Earlier, strings were likened to quoted speech. Quotes are used when repeating exactly what was said, but there is no necessity to understand the words quoted—they could be nonsense.

This is the principle behind the idea of strings. The computer doesn't understand what they are or what they mean. It just treats them as meaningless symbols.

Anything in a program which is placed between pairs of double inverted commas will be regarded as a string by the computer. Here is a string:

"How old are you?"

Those same characters could appear in a book as reported speech:

"How old are you?", said Jim.

Notice also that you can put numbers into quoted speech:

"I'm 22", said Sally.

Try these examples:

```
PRINT 2*2
```

This example tells the computer to do a small sum: put *the value* of 2 multiplied by *the value* of 2 on the screen—i.e. do the sum and display the result.

```
PRINT "2*2"
```

This example looks almost exactly the same as that above it. But the inverted commas make 2*2 into a string. We are now telling the computer to put 2, then *, then 2 onto the screen.

The computer can tell you how long a string is, if you ask it to. Try this program:

```
100     LET A$="COMPUTER"
110     LET B$="MICROCOMPUTER"
120     !
130     !     Lines 100 and 110 declare two string
140     !     variables, A$ and B$. String variables
150     !     are just like numeric ones, except that
160     !     they are names for groups of symbols,
170     !     not for the values of numbers. The name
180     !     given to a string variable must always
190     !     have a dollar sign ($) at the end of it.
210     !
220     LET A=LEN(A$)
230     LET B=LEN(B$)
240     !
250     !     A and B are numeric variables which
260     !     contain the lengths of the two strings.
270     !     LEN tells you the number of characters
280     !     in a string (i.e. its length).
300     !
310     PRINT "The string called A$ is ";A;
        " characters long."
320     PRINT "The string called B$ is ";B;
        " characters long."
330     END
```

Try changing the contents of A$ and B$—the computer will always be able to tell you how many symbols each one contains.

SPACES

A space may be nothing to you, and it's understandable if you think it doesn't count. But the computer treats it as though it were a symbol. Try changing A$ and B$ (in the program above) to:

"   "   and   "   "

You will see then that the computer tells you how many spaces they contain, although it looks as if both are empty. A truly empty string would look like this:

""

As you can see, it doesn't even contain spaces. This is called a *null string*.

STRINGS INSIDE STRINGS

The Enterprise can do some interesting things with strings. For instance, you can make one string out of another one. Here's a program which does this:

```
100   LET BIGSTRING$="quite a few words"
110   !
120   !    Line 100 declares a string variable.
130   !
140   LET SMALLSTRING$=BIGSTRING$ (9:17)
150   !
160   !    Line 140 declares another string
170   !    variable—with a difference. It starts at
180   !    the 9th character of BIGSTRING$ and
190   !    ends at the 17th character.
210   !
220   PRINT BIGSTRING$
230   PRINT SMALLSTRING$
240   END
```

The computer literally copies some characters from one variable into another separate variable. This separate variable is called a substring. To make one string out of another, we have given the substring a name—SMALLSTRING$—and then specified, in brackets after the name of the larger string, the character where we wanted the substring to start and the character where it would end.

So a statement such as LET INITIAL$=NAME$(1:1) says 'call the substring INITIAL$ and let it contain the first letter of NAME$'.

In each of these examples we have declared our substring as a variable in its own right, by giving it a name (an 'identifier') of its own. This isn't strictly necessary, though. In the above program, we could delete line 140 and change line 230 to:

```
230   PRINT BIGSTRING$ (9:17)
```

—meaning 'print the ninth to seventeenth characters of the string called BIGSTRING$'. The method you choose will depend on the reason why you're using substrings. If you want to use the same substring several times, it's often more convenient to declare it as a separate variable. If not, then the second method above will suffice.

INKEY$

INKEY$ is a very useful and important string function. (Any BASIC word which does something to a string is called a string function; most of them end with the $ sign, to indicate that the result they produce is also a string.)

INKEY$ allows you to press a key while the program is running, so as to affect the way that the program will continue. In some ways it's like an INPUT, but the differences will soon become clear. Try this:

```
100     PRINT "Have you understood this chapter?"
110     PRINT
120     PRINT "Answer with y or n"
130     DO
140             LET A$=INKEY$
150     LOOP UNTIL A$<>""
160     !       ______________________________
170     !       Line 140 puts the result of a key-
180     !       press into a string variable. You'll
190     !       see why this is important.
200     !       ______________________________
210     IF A$="y" THEN
220             PRINT "Let's hope you're right."
230     ELSE IF A$="n" THEN
240             PRINT "Well, read it again."
250     ELSE
260             PRINT "That was a wrong key."
270     END IF
280     !       ______________________________
290     !       You have seen IF/THEN statements
300     !       before. The purpose of ELSE in lines 230
310     !       and 250 is fairly obvious from the
320     !       ordinary meaning of the word. 'Con-
330     !       ditional' statements like these are dealt
340     !       with at length in the chapter on decisions.
350     !       ______________________________
360     !       END
```

When the program asks its question, your next keystroke gives the answer and makes the computer proceed to line 21Ø and onwards.

Unlike an INPUT statement, the INKEY$ function only asks for one key to be pressed, and doesn't require you to use 'enter'. (So it's very useful for games—for one-letter answers, or for continuing an action that has been stopped in the middle by a loop.) Note that 'shift' plus another key counts as just one keystroke.

Another difference between INKEY$ and INPUT is that INKEY$ doesn't automatically make the computer wait until your answer has been given. That's why lines 13Ø and 15Ø in the above program are necessary. Try deleting them. You will then see that the program runs right through to the end in a fraction of a second, giving you no time to type your answer. When the computer comes to line 14Ø, it decides that since none of the keys has just been pressed, INKEY$ is equivalent to a null string (" "); so it makes A$ into a null string too.

Lines 13Ø and 15Ø, on the other hand, say 'keep looping until A$ becomes something *different* from a null string—*then* go on to the rest of the program'.

Bear in mind that INKEY$ is like a variable of which the value changes very quickly. The point is that the computer 'looks' at the keyboard approximately once every fiftieth of a second, to register any key-press made in that interval; if you press (say) the 'a' key, then INKEY$ becomes equivalent to "a"—but only for a very short time. Normally, one-fiftieth of a second later, INKEY$ will revert to a 'null' string, and will remain with this value until your next keystroke.

This explains why line 14Ø is needed in the program. It immediately puts the result of your keystroke into a separate string variable which will not change while the computer is examining it (testing it to see if it meets certain conditions—see lines 21Ø to 27Ø).

You could try deleting line 14Ø and altering lines 15Ø, 21Ø and 23Ø to:

```
15Ø     LOOP UNTIL INKEY$<>" "
21Ø     IF INKEY$="y" THEN
23Ø     ELSE IF INKEY$="n" THEN
```

The reason why the program doesn't work properly any more is that the value of INKEY$ changes (it reverts to " ") in between lines 15Ø and 21Ø.

There is, of course, no need to put the value of INKEY$ into a separate variable if it is not going to be used in subsequent program lines. For example, if you merely want to hold up the action of a program until the user tells it to continue, you can type something like:

```
100     PRINT "Press any key to continue."
110     DO
120     LOOP UNTIL INKEY$< >" "
```

UCASE$ AND LCASE$

Another interesting thing you can do with strings is to change them into capital or small letters.

The two words which will do this are UCASE$ and LCASE$. This program will show you how it's done:

```
100     !   ----------------------------------------
110     !   This program will convert a string that
120     !   you have typed, first into capitals and
130     !   then into small letters.
140     !   ----------------------------------------
150     INPUT PROMPT "Please type in some letters:
        ": A$
160     PRINT A$
170     PRINT UCASE$ (A$)
180     PRINT LCASE$ (A$)
190     END
```

VAL

Something else which you can do is to convert a string which contains number-*symbols* into the number that it would be if it were not a string. The BASIC word which does this is VAL (short for 'value'). Try this:

```
100     INPUT PROMPT "Please type in some
        characters: ":A$
110     PRINT A$
120     PRINT VAL(A$)
130     END
```

To determine VAL(A$), the computer will only count the numbers in A$ which come before the first letter. So VAL("123AB45") equals 123, while VAL ("AB12345") equals Ø.

You can also do the opposite—convert a number into a string—by using the word STR$.

ADDING STRINGS TOGETHER

You have already seen how you can turn part of a string into a substring. Another way to form a new string is by linking up strings (or portions of strings) to one another. The act of joining strings together is called concatenation; all you do is put an ampersand ('&') between the strings—as if to say, 'one string *and* another string'.

The following example makes use of concatenation in addition to substrings and the LEN function.

```
100     INPUT PROMPT "Please type a word:
        ":STRING$
120     CLEAR SCREEN
130     LET ABBREV$=STRING$ (1:1) & STRING$
        (LEN(STRING$): LEN (STRING$))&"."
140     !
150     !    In line 130, STRING$(1:1) makes the first
160     !    character of STRING$ into a substring.
170     !    LEN(STRING$) gives the number of
180     !    characters that STRING$ contains, so
190     !    STRING$ (LEN(STRING$): LEN
200     !    STRING$)) makes another substring out
210     !    of the final character. Then, using the '&'
220     !    symbol, the two substrings are linked to
230     !    each other and to a full stop, thus
240     !    defining a new string variable,
245     !    ABBREV$.
250     !
260     PRINT STRING$;" abbreviated is "; ABBREV$
270     END
```

You would do it this way if you wanted to use the abbreviation repeatedly. If you were only using it once, you could of course delete line 130 and change line 260 to:

```
260     PRINT STRING$ (1:1) &STRING$
        (LEN(STRING$): LEN(STRING$))&"."
```

In this case the ampersands could be replaced by semicolons.

LOOPS

You have already seen some short programs which use loops to make the computer repeat itself. Should you wish the machine to perform one operation several times in succession, a loop represents a much easier way of doing this than typing in the same piece of BASIC over and over again.

A loop is also an easy way to control an entire program, as you will see.

With one exception (GOTO, which can be treated as a loop), every loop contains a special statement to mark its beginning, and ends with a line that tells the computer to jump back to the start again.

DO/LOOPS

Let's deal with the easiest kind first, DO/LOOPs. You have already come across these. They look like this:

```
 80      INPUT PROMPT "Please type in a number
         and I'll print the 'times table' for it. ":A
 90      LET B=A
100      DO !          Beginning of DO/LOOP
120        LET B=B+A
130        PRINT B,
140        !
150        !         Line 90 puts your number into
155        !         another variable, B. Then, in
160        !         lines 120 and 130, the value of
170        !         A (which has not changed) is
175        !         added to B, and the new value
180        !         for B is put on the screen. This
185        !         happens each time round the
190        !         loop, so B increases by the value
200        !         of A each time.
210        !
220      LOOP UNTIL B > 150 ! End of DO/LOOP
230      !
240      !           UNTIL B > 150 stops the program
245      !           going on forever without any
250      !           interference from you (e.g. 'stop' or
260      !           'hold' keys). UNTIL means exactly
265      !           what you might think it does—i.e.
270      !           'until B is bigger than 150'.
280      !
290      END
```

A DO/LOOP, as you can see, makes the computer 'go around in a circle' until some condition is met which tells it to stop. This condition may be specified by using the word UNTIL or WHILE.

Whichever of these words is used, you have a choice between DO WHILE (or DO UNTIL) and LOOP WHILE (or LOOP UNTIL). Usually, either alternative will be possible, particularly in a program like the previous one, which is not dependent on an especially delicate arrangement of conditions.

The essential difference between putting a condition at the beginning and putting it at the end of a loop is that it affects, by a difference of 1, the number of times a loop may happen. If the condition is at the beginning, it will be tested before the loop is done, which means that if it is met immediately the loop will never be done. If it is put at the end, the condition cannot actually be read by the computer until then, which means the loop will always be carried out at least once. WHILE and UNTIL are opposite types of conditions. UNTIL B= 15Ø and WHILE B=15Ø mean totally different things (UNTIL means 'as long as it is not' and WHILE means 'as long as it is').

Try changing the loop above by using a WHILE condition in place of UNTIL, by using bigger and smaller numbers in response to the INPUT PROMPT at the beginning, and by putting the condition after the DO instead of after the LOOP. Experimentation will soon show you what you can do with a DO/LOOP.

GETTING OUT

At any time, if you want to, you can make the machine get out of a loop by using the word EXIT. Now, EXIT applies to FOR/NEXT loops as well (these are described opposite). So you need to specify which type of loop you are exiting—EXIT FOR or EXIT DO is the way to do this. You should exit using some sort of condition—e.g.:

```
1ØØØ     IF X > 25 THEN EXIT DO
```

A quick look at the chapter about decisions will tell you how to do this using SELECT CASE as well.

Remember that the word EXIT is the only proper way to leave a loop before it should end. Things like GOTO should not be used for this purpose—they can be confusing both for you and for the computer. (GOTO and its associated command GOSUB are dealt with on pages 127-128). EXIT will simply make the computer jump to the first line after the end of the loop.

FOR/NEXT LOOPS

FOR/NEXT loops are rather different from DO/LOOPs. They are possibly a little less straightforward, but they have definite uses.

Here is a short FOR/NEXT loop:

```
100     INPUT PROMPT "How many times would you
        like me to loop? ":X
110     !
120     !
125     !
130     !
140     !
150     !
160     FOR P=1 TO X
170        PRINT "LOOP THE LOOP!"
180     NEXT P
190     END
```

Notice that in this and other INPUT PROMPTs, a space is put at the end of the prompt itself. This makes the screen look tidier.

Do you see how the number you type is the number of times the computer will perform the loop? This is the essential purpose of a FOR/NEXT—you can specify quickly and concisely how many times the loop should be done. There is no need to use conditions except in special circumstances.

FOR/NEXT can also count in successions of three, twenty, 0.2, 1000, 466.666 or even backwards. STEP is the key to this little ability. Here's a program which counts backwards by twos.

```
10      FOR P=40 TO 0 STEP -2
20         PRINT P,
30      NEXT P
40      END
```

You cannot simply ask the machine to count from 40 to 0 without giving it a STEP command. It has to know to subtract (counting in minus numbers) from the number given in the FOR line before it will do this. Unless you specify by using STEP, a FOR/NEXT loop will always count upward in steps of 1. Note also the PRINT P statement. The first line of the loop creates a variable, in this case P, of which the value changes each time the loop is performed. The 'times table' program, used as an example of the DO/LOOP, will be shorter if we use FOR/NEXT:

```
100     INPUT PROMPT "Please type in a number
        and I will give you a 'times table' for it. ":A
110     FOR P=0 TO 150 STEP A
120        PRINT P,
130     NEXT P
140     END
```

That program does almost the same thing as the DO/LOOP (page 59) but in a different way. DO/LOOPs are just a little easier to read and understand when you're unfamiliar with computer programs.

Notice that the FOR/NEXT loop is always tested at the beginning, and will not run if it has gone past the end limit specified in the FOR statement. Try entering numbers bigger than 150 for the 'times table' program above.

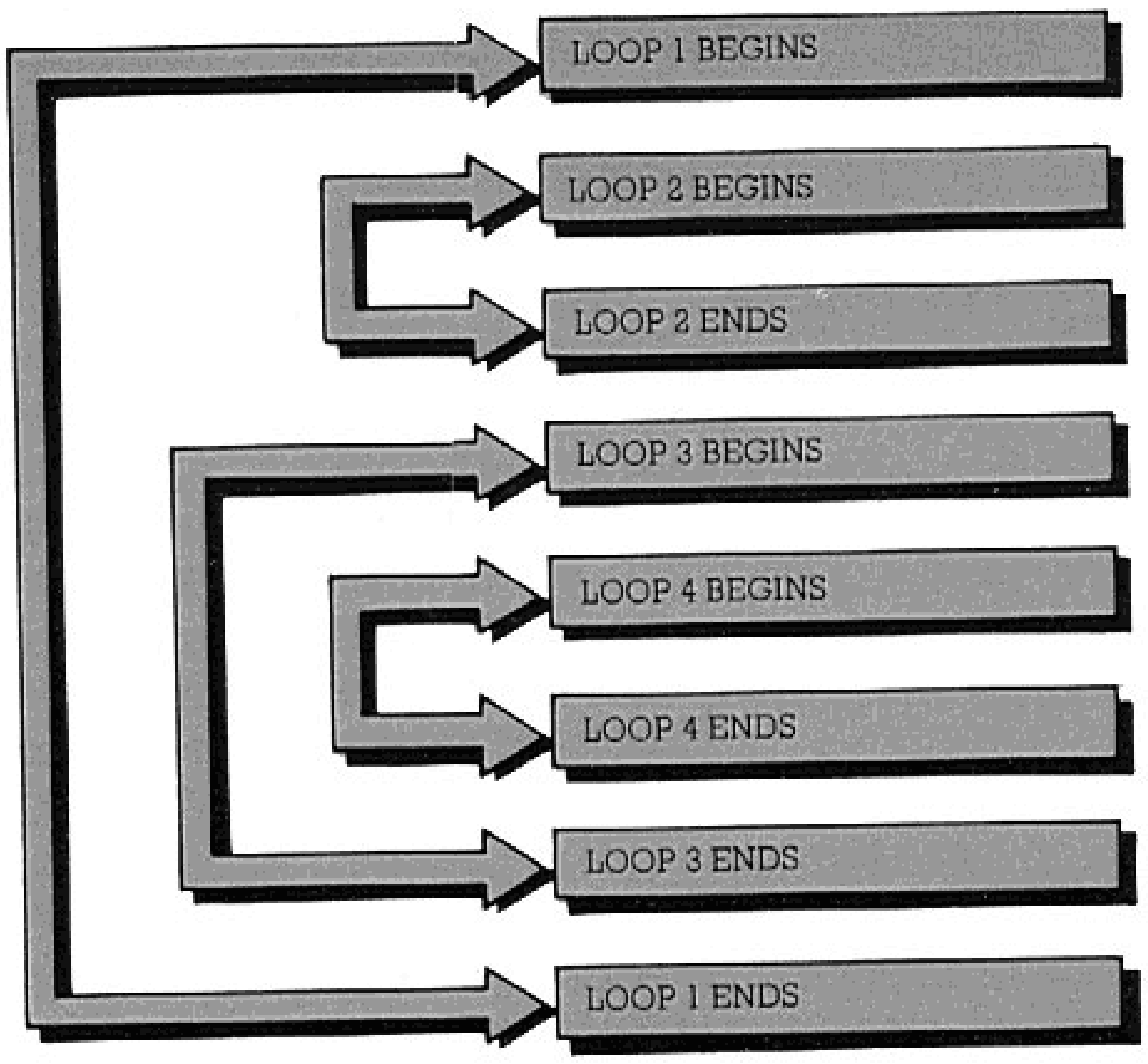
You might like to try different ways of presenting this program using PRINT AT (page 31). You could, by combining PRINT and GRAPHICS commands (page 89), design a table made up of boxes for your 'times table' to go in. And this time, the manual isn't going to tell you any more! Programming is all about being adventurous and finding things out for yourself. You may have some better ideas, as well. If you're a parent with young children or you have a little brother or sister, this is the kind of program which could be used to teach them and to have a bit of fun as well.

NESTED LOOPS

These words might sound rather strange. Nested loops, though, are just loops placed inside loops. The important thing to remember is never to end the second loop outside the first one. Here is an example:

```
100     FOR P=1 TO 20 !          First loop begins.
110        PRINT P,
120        FOR A=1 TO 4 !        Second loop begins.
130           PRINT P+10; P-10;
140        NEXT A !              Second loop ends.
145        PRINT
150     NEXT P !                 First loop ends.
```

You can nest another loop inside the second one as well, and another inside that, and so on. If you look at a series of nested loops, the end lines should come in descending order as you look down the program. The beginnings should be in ascending order. Look at the diagram.



As you can see from the diagram, if you really felt that loops were the answer to all your problems, you could also sandwich loops in between two beginnings or two endings of other loops.

However, complex arrangements of loops like this need presence of mind if you are to keep track of them. The Enterprise helps by indenting loops as shown in the last program, which makes them easier to read. Comment lines also help to make things more easily understood. Normally, if you have several layers of nested loops, it is easier to define the inner 'layers' as functions (see page 79). This keeps the purpose and flow of the program clear.

DECISIONS, DECISIONS!

Fiction often refers to computers with the reasoning of human beings. Perhaps, like most people, you believed before you understood computers that they are more 'clever' than they really are.

But while they can't think like people can, computers can make very simple decisions and comparisons.

COMPARISONS USING IF/THEN

Perhaps we can start with comparisons. Do you remember the relational operators mentioned in part 1 of the manual? These are the key to the way in which computers make comparisons and, sometimes, decisions based on those comparisons.

Try out this program. It will show you how to use these relational operators to decide whether two strings are different or not.

```
100    LET THAT$="kettle"
110    INPUT PROMPT "Please type in some letters:
       ": THIS$
120    !  ------------------------------------------
130    !            100 and 110 should be familiar.
140    !            The two strings are compared in
150    !            220 and 230, using < > which
155    !            means 'different from'. If the two
160    !            are the same, both will be printed.
170    !            If they are different, "kettle" is
180    !            printed once.
190    !  ------------------------------------------
200    PRINT
210    PRINT
220    IF THIS$=THAT$ THEN PRINT   THIS$;" = ";
       THAT$
230    IF THIS$< >THAT$ THEN PRINT THAT$
240    !  ------------------------------------------
250    !            220 and 230 use IF/THEN to make a
260    !            decision. IF THIS$ is the same as
265    !            THAT$ THEN put both of them on
270    !            the screen. If it is not the same,
290    !            print THAT$.
295    !  ------------------------------------------
300    END
```

IF/THEN is one of the two main ways in which the Enterprise can make a decision. It is one of the more English-like statements in BASIC. It is used with logical operators to look at variables and numbers to see if they fulfil certain conditions. IF/THEN is not used to perform calculations on variables, but it is there to redirect a program IF something about a number or string is true.

For instance, you may write a program to play a game in which each player only has a certain number of turns per round of the game. So you set aside a variable which goes up by 1 every time a player has completed a turn.

Then, when the variable is equal to the number of turns the player should be allowed, an IF/THEN statement can be used to tell the program to set the variable back to Ø for the next player, inform the current player that his round is over, record his score and go on to the next player, etc.

IF BLOCKS

An IF/THEN statement does not have to be just one line long. You can also use what is called an IF block. This is simply several lines which allow the computer to make several comparisons or decisions. It also means you can use several lines to deal with each condition. That gives you great scope for making decisions.

Lines 22Ø and 23Ø of the program opposite could have been written in a different way. The lines below could be used to replace 22Ø and 23Ø in the original program. They would have the same effect.

```
22Ø     IF THIS$=THAT$ THEN
224             PRINT THIS$;" = ";THAT$
23Ø     ELSE
232             PRINT THAT$
236     END IF
```

The reason is that the computer had just two alternatives. Either the two strings were the same or they were not. So the lines above say: 'If THIS$ and THAT$ are the same, print them both. If anything else is true, print THAT$ only.' This is an example of a small IF block—it contains one IF statement, one THEN statement, one ELSE statement (you can only put one ELSE statement in an IF block, for obvious reasons), and the words END IF, which let the computer know it doesn't have to make any more comparisons.

ELSE means exactly what it says. Anything else. It means you can use IF/THEN to tell the computer to do particular things if so-and-so is 'true' and then use ELSE to cater for any other possibilities. It must always be on a line of its own—never on the same line as an IF/THEN statement.

The example below is an IF block. It also uses a numeric function that you have seen once before (pages 5-6).

```
100     RANDOMIZE
110     LET A=RND (5)
120     !
130     !       100 and 110 make numbers out of the
140     !       blue! These can be used to make
150     !       patterns on the screen, to play games
160     !       like roulette or Simon (the flashing
170     !       sequences game) or many other things.
180     !       Here we'll use the random numbers to
190     !       draw shapes on the screen. The
200     !       commands are explained in the chapter
210     !       on graphics. The word RND is the one
220     !       which makes the computer produce a
230     !       random number. RANDOMIZE makes
240     !       sure the number is different each time
250     !       the program is run.
260     !
270     GRAPHICS
280     PLOT 600, 300;
290     OPTION ANGLE DEGREES
300     PLOT ANGLE 0;
310     IF A=1 THEN
320        PLOT FORWARD 100;
330        PLOT LEFT 90;
340        PLOT FORWARD 100;
350        PLOT LEFT 90;
360        PLOT FORWARD 100;
370        PLOT LEFT 90;
380        PLOT FORWARD 100
390        !
400        !    270 to 300 prepare the computer for line
410        !    drawing. In this example, some very
420        !    simple graphics commands are used.
430        !    LEFT gives turns; FORWARD is followed
440        !    by a number of screen positions
450        !    (measured according to the graphics
460        !    conventions). Lines 320 to 380 draw a
```

```
465       !   square.
470       !   ____________________________________
480     ELSE IF A=2 THEN
490       PLOT FORWARD 50;
500       PLOT LEFT 90;
510       PLOT FORWARD 100;
520       PLOT LEFT 90;
530       PLOT FORWARD 50;
540       PLOT LEFT 90;
550       PLOT FORWARD 100
560       !   ____________________________________
570       !   490-550 draw a rectangle.
580       !   ____________________________________
590     ELSE IF A=3 THEN
600       PLOT FORWARD 100;
610       PLOT LEFT 120;
620       PLOT FORWARD 100;
630       PLOT LEFT 120;
640       PLOT FORWARD 100
650       !   ____________________________________
660       !   600-640 draw a triangle.
670       !   ____________________________________
680     ELSE
690       TEXT
700       PRINT "I have no instructions for number"; A
710     END IF
720     END
```

Now you've had a chance to play with the Enterprise's graphics, let's return to IF/THEN. As you can now see, you can use all these numbers and BASIC words to help you draw pictures—and also to make music and sound effects. All you need to do is combine 'ordinary' BASIC with the graphics commands.

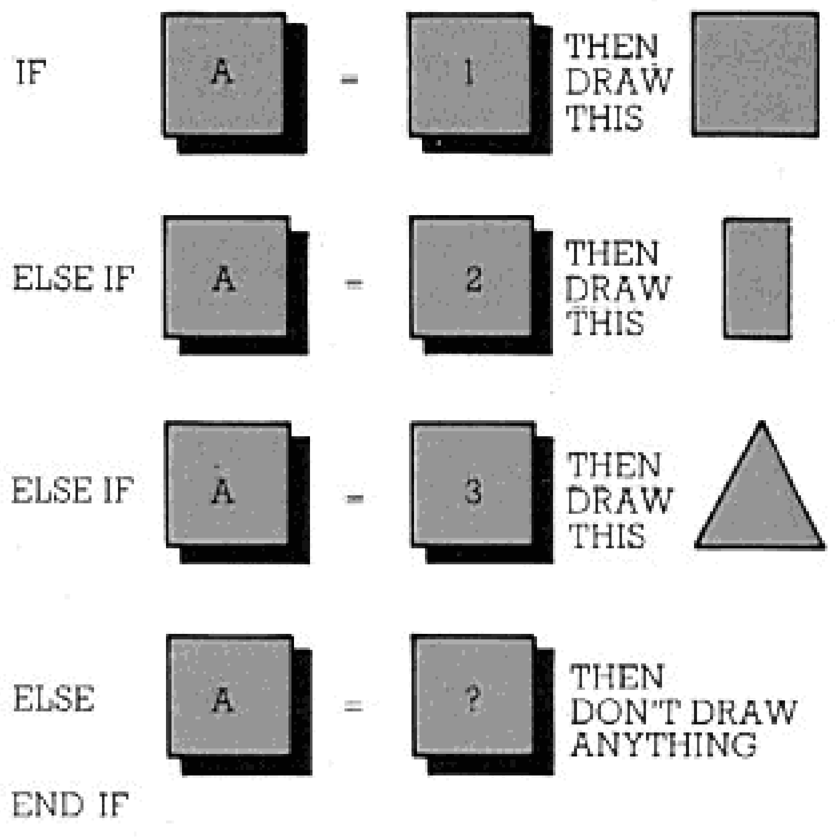
The program above makes its own random number by using a formula (you'll never need to worry about how, unless you get really interested in the inner workings of the computer) to produce numbers 'out of thin air'. Incidentally, if you wanted to, you could print a sequence of these random numbers by using a DO/LOOP, and you would not be able to see any pattern in the sequence.

Then, depending on the number produced by RND, the computer will go off and draw one of three possible shapes. If the number is not one which is mentioned in an IF line, it will instead run the solitary

ELSE statement (line 68Ø) and display the message in line 7ØØ—which is really a bit miserable compared with all the rest of the program. This is what was meant by using IF/THEN/ELSE to put together more complex conditions by making them into several lines—a block—rather than one or two.

The previous program shows exactly how you should use an IF block. IF and ELSE IF separate each possible course of action.

An IF block *must* always finish with END IF—otherwise the computer will not run the rest of the program if the last IF test failed.



IF/THEN is, therefore, good as one line to pick out exceptions or deal with decisions when only something simple needs to be done as a result. As a block it can be used to run an entire complex program, depending on a decision which is made through the IF block at the beginning.

SELECT CASE

With SELECT CASE, you can make very similar comparisons and decisions to those offered by IF/THEN. Think of CASE as meaning 'in case of...', rather than 'if...then...or else...'.

In the example which follows, notice the lack of the variable name in the CASE statements. CASE 1,2,3 is right, CASE X=1,2,3 is wrong—the variable to be tested has already been given by the SELECT line. Using CASE to make one decision out of several alternatives is, as you can see, shorter and clearer than using IF/THEN.

Just as with END IF, you must mark the end of the SELECT block—in this case by typing END SELECT.

```
100   CLEAR SCREEN
105   !
110   !     Line 100 clears the screen. Then
111   !     a menu is printed in the middle
115   !     of the screen, using PRINT AT to
118   !     position the letters. After that,
120   !     you are asked to type in your
122   !     choice, and the number you choose
125   !     is put into a variable called A.
127   !     If you type in a number bigger
130   !     than 3 or less than 1, the program
133   !     just asks you to do it again.
135   !     410 deals with that using IF/THEN.
136   !     It also refuses to allow numbers
137   !     with a decimal point. A <> INT(A)
138   !     makes this possible.
140   !
150   PRINT AT 9,18:"MENU"
200   PRINT AT 11,10:"1) Print my name"
250   PRINT AT 12,10:"2) Print your name"
300   PRINT AT 13,10:"3) Print both of our names"
360   DO
400         INPUT AT 16, 10, PROMPT "Please enter
            your choice: ":A
410   LOOP WHILE A < 1 OR A > 3 OR A <> INT(A)
450   CLEAR SCREEN
460   !
465   !     The program from Lines 500 to
466   !     1000 makes the decision about
467   !     what to do with your choice, and
468   !     does it. The numbers after the
470   !     word CASE each time are the
472   !     numbers which A can be equal to.
473   !     For instance, 550 and 600 say,
474   !     'In the case of 'A' being 1,
475   !     print "ENTERPRISE!!!" in the
476   !     middle of the screen.' 1000 is
477   !     essential to tell the machine no
479   !     more cases can be expected.
480   !
500   SELECT CASE A
550   CASE 1
600     PRINT AT 9,18: "ENTERPRISE!!!"
```

```
650    CASE 2
700      INPUT PROMPT "Then please tell me your
         name. ":NAME$
750      PRINT AT 9,18:NAME$
800    CASE 3
850      INPUT PROMPT "Then please tell me your
         name. ":NAME$
900      PRINT AT 9,18:NAME$
950      PRINT AT 11,18: "Enterprise"
1000   END SELECT
1010   !  ------------------------------------------
1015   !      Lines 1200 to 1350 are very similar
1016   !      to the check on 'A' at the
1020   !      beginning of the program. They
1022   !      make sure that the program will
1025   !      only end or go back to the
1027   !      beginning if the first letter of
1030   !      A$, converted to a capital, is
1035   !      either "Y" or "N". GOTO 100 tells
1037   !      the computer to jump back to the
1040   !      beginning of the program only if
1041   !      the first letter of A$ (converted
1043   !      to a capital) is "Y". Otherwise
1045   !      the program ends—which, as you
1046   !      can see, it can only do if the
1047   !      first letter of A$ is "N".
1048   !  ------------------------------------------
1050   PRINT
1100   PRINT
1150   PRINT
1200   DO
1250            INPUT PROMPT "Would you like to
                do that again? ":A$
1300   LOOP UNTIL UCASE$(A$(1:1))="Y" OR
       UCASE$(A$(1:1))="N"
1350   IF UCASE$(A$(1:1))="Y" THEN GOTO 100
1400   PRINT
1450   PRINT
1500   PRINT
1550   END
```

Try writing a program to simulate a dice. You can do this using either IF/THEN or SELECT CASE. Use a random number as in the program on page 66. This, of course, would have to be between 1 and 6. So

```
RND (6)+1
```

would take care of that. Knowing you have six possible results, you shouldn't find it hard to do. Perhaps you could use graphics to put a picture of the dice on the screen.

Remember that using IF/THEN as an IF block does not mean you have to use ELSE. ELSE lines are optional, just like the number of possibilities mentioned within the block or the number of CASE selections.

CASE can also have an 'ELSE clause'. This is CASE ELSE—just like CASE 1 or CASE "HELLO". Notice that you can use CASE with strings as well, and that an advantage over IF/THEN is the ability to lump several 'truths' together with the same result—like this:

```
CASE 1,3,5
PRINT "These are odd numbers."
```

Lastly, experiment with ASCII codes (see 'The Character Set', page 104). It is possible to use these to get a computer to put strings into alphabetical order. ASCII codes are just numbers which stand for characters. The use of arrays (which come next) is also helpful in this sort of programming task.

As you've discovered, programming is chiefly a way to handle information. Until now, you have only used small amounts of information in programs, either sentences and numbers that you have typed in response to INPUT statements, or sentences the computer has displayed for you to read.

You've probably wondered how you would go about dealing with larger amounts of information—a long list of words or numbers, maybe, or several paragraphs of instructions for a game or for a program which helps you work out your finances. The Enterprise provides some very efficient ways to manipulate lists of numbers or large groups of words.

The way in which you would keep a list of names, for instance, in a program would be to use an *array*. An array is like a big variable which can contain several smaller variables. We began by comparing variables to boxes, the contents of which may change. An array can be thought of as a large box in which a certain number of small boxes are stored. Alternatively, you can visualize it as a page from a notepad on which you can put a list.

NUMERIC ARRAYS

As with variables, you have string arrays and numeric ones. So to declare a numeric array you would use (e.g.):

```
NUMERIC STORE (1 TO 10)
```

Type this into the computer as line 100. It tells the computer to set aside a 'container' called STORE with space for 10 smaller variables in it. These variables are known as the *elements* of the array, and you can record whatever numbers you like in these little spaces. The seventh variable (or element) inside STORE would be called STORE(7).

Now type in the following lines:

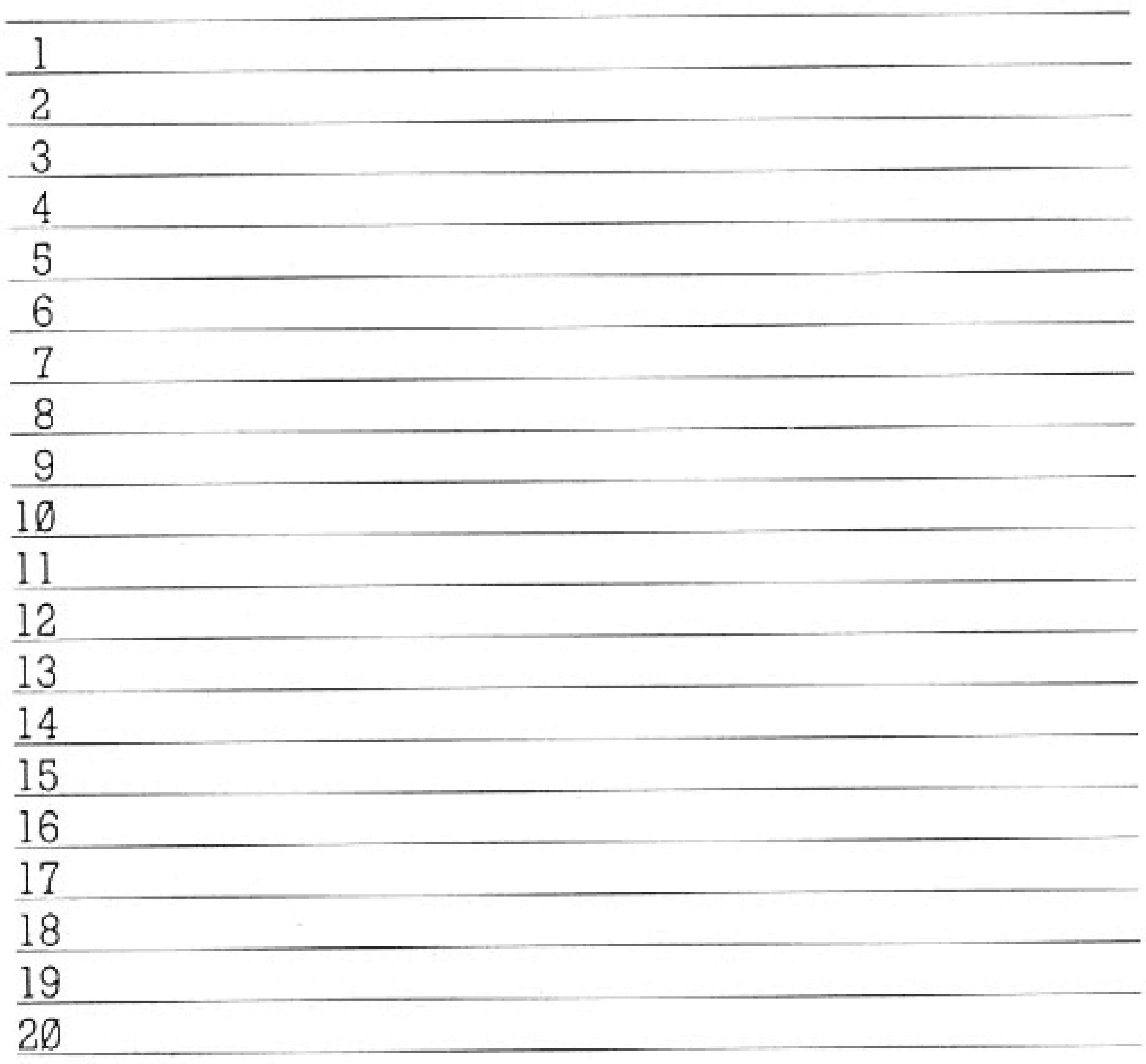
```
110  FOR S=1 TO 10
120      INPUT PROMPT "Enter a number ": STORE(S)
130  NEXT S
```

This enables you to put numbers into your array. Run this, and when it has finished, type in PRINT STORE(9) or (8) or any other number from 1 to 10. As you can see, you can call up numbers from the array. You've put a list of numbers into the computer. If you wanted to record the temperatures each day for the month of December, you could do this in an array called DECEMBER with elements numbered 1 to 31. The temperature for each day would form the contents of an element, and the element numbers would signify the dates.

An array could also consist of elements numbered 56 to 76, or 123 to 171, or even 12345 to 12445. If you like, a *negative* number can be used for the upper or lower end of the element range—e.g. NUMERIC TABLE (-1Ø TO 1Ø), or NUMERIC TABLE (-2Ø TO -1Ø). Element numbers are just for reference.

TWO-DIMENSIONAL ARRAYS

An array can be more complex than just a list. The diagrams below and on the next page show you the difference between one-dimensional and two-dimensional arrays.



A two-dimensional array can be pictured as a grid. Or (if you like), you can imagine that this time you are storing your boxes in a cupboard where each shelf is given a number and there is room for the same number of boxes on each shelf. A (1 TO 4, 1 TO 4) would produce an array which can be visualized like this:-

1.1	1.2	1.3	1.4
2.1	2.2	2.3	2.4
3.1	3.2	3.3	3.4
4.1	4.2	4.3	4.4

An example of the use of this kind of array is as the board for a game like draughts or Othello. The array could be used to store information as to which pieces are on which squares of the board. You could also use an array like this as a table of numbers or words—a more versatile sort of list.

STRING ARRAYS

String arrays are declared in the same way as numeric ones, except that the word STRING is used. With string arrays it is also possible to alter the length of each element— this cannot happen in a numeric one.

Altering the length of elements in an array just means altering the largest amount those elements can hold. Just as you aren't forced to fill a bucket to the brim every time you put water in it, so you don't have to fill the elements in an array completely.

As you already know, the Enterprise has a certain amount of space inside it, in which it can store programs and information to be used with programs.

When you declare a string array the computer will set aside a certain amount of space for each element. This is called MAXLEN. MAXLEN is, unless you specify otherwise, 132 characters. This does not affect numbers; it only relates to strings.

Should you wish to conserve space in your computer's memory—or to allow for longer elements than 132 characters—you must tell the computer so. This is how:

```
STRING ARRAY$(3Ø TO 5Ø)*1Ø
```

That BASIC statement would set aside an array with 2Ø elements in it (numbered 3Ø to 5Ø), each element being 1Ø characters long. MAXLEN(ARRAY$(45)) would then be 1Ø. The longer your elements are, the more you can put into them, but they will take up more memory and leave less space for other things. So you should only make them longer than normal if it is really necessary, or if you are certain you will have enough space for them. If you expand your Enterprise's memory, you will naturally be able to store much bigger programs and put much more in your arrays.

DECLARING VARIABLES

Simple variables can also be declared using STRING or NUMERIC. NUMERIC A declares a numeric variable. NUMERIC A,M,N,H,D or something similar could be used to declare all your numeric variables at once—at the beginning of a program. STRING A$,B$,HELLO$*8... would do the same with string variables. The only difference between declaring variables like this and declaring an array is that, in the case of an array, elements are specified.

READ/DATA

Many BASIC words go hand in hand with other BASIC words. READ and DATA (and the word RESTORE as well) are such words.

These are mentioned in this part of the manual because they represent yet another way to keep large amounts of information inside a program. They also represent another way to put numbers or strings into an array. READ is the word which does all the hard work. Here is a short example.

```
 8Ø     LET P = 1
 9Ø     DO UNTIL P = Ø
1ØØ         READ P
11Ø         PRINT P,
12Ø     LOOP
13Ø     DATA 1,2,3,4,5,6,7,8,9,1Ø
15Ø     DATA 11,12,13,14,15,16,17,18
16Ø     DATA 19,2Ø,21,22,23,24,25,Ø
17Ø     END
```

The DATA statements simply hold the information. Each item of information must be separated from the next with a comma—this indicates that one piece of information is over and the next one follows. READ tells the computer to look at one DATA item and do with it whatever you want. In this case, the item is placed in the variable P, and then the contents of P are printed on the screen.

Notice that the same variable name is used for all the DATA items. This does not matter—the same one is being re-used. So the program READs one DATA item, puts it into the variable called P, prints it on the screen, READs the next, puts that into P instead of the last DATA item, and so on. The Ø at the end of the last DATA statement is used to tell the computer no more DATA follows.

The computer will only read each DATA item once. Once an item has been read, the machine 'remembers' its position and goes on to the next one (reading from left to right and top to bottom as you would read a book). When it's read them all, it considers there are no more there—and will be confused if you try to tell it to go on looking!

The program below demonstrates the use of READ/DATA with strings.

```
15Ø    CLEAR SCREEN
2ØØ    PRINT "I'm going to tell you a story."
25Ø    PRINT
3ØØ    PRINT "Here goes!"
35Ø    PRINT
4ØØ    PRINT
41Ø    FOR X = 1 TO 5ØØØ
415    NEXT X
42Ø    !  ________________________________
43Ø    !      Lines 41Ø and 415 merely specify an
```

```
440   !     interval. The DO/LOOP (550-810 is the
450   !     important part. Let's go round it and see
460   !     what happens. First it READs A$—which
470   !     means it looks for a DATA statement
480   !     (which can be anywhere in the program),
490   !     reads it and checks to see if
500   !     it's "END". If it is, the computer
515   !     stops looping and goes to
520   !     the rest of the program. If not, 700
525   !     tells the computer to print the
530   !     string from the DATA statement
535   !     on the screen (and put a space after it).
540   !     ------------------------------------------
550   DO
600         READ A$
650         IF A$="END" THEN EXIT DO
700         PRINT A$;" ";
750         FOR Y=1 TO 500
760         NEXT Y
810   LOOP
820   !     ------------------------------------------
830   !     The rest of this program may look
840   !     confusing. If you remember that the
850   !     DATA statements are read by an
855   !     earlier part of the program, and that
870   !     900-1100 are actually performed
873   !     after the DATA has been read and
875   !     put on the screen, the logic of
880   !     the program will fall into place.
890   !     ------------------------------------------
900   PRINT
950   PRINT
1000  PRINT "                              THE END"
1050  PRINT
1100  PRINT
1125  END
1150  DATA Once,upon,a,time,there,was,a,
      little,computer,
1200  DATA called,the,Enterprise.,It,was,
      a,very,
1250  DATA happy,computer.,All,the,best,
      programmers,in,
1300  DATA a,land,called,England,had,worked,
      all,day,
1350  DATA and,all,night,for,months,to,make,
      the,Enterprise,
1400    DATA the,best,computer,ever.,Today,
        you're,learning,
1450    DATA to,write,BASIC,on,the,Enterprise.,
        Aren't,you,
1500    DATA lucky?,END
```

There is a possible modification you could make to this program. Try adding 100 DO 1120 LOOP to this program. Then RUN it again. As you will now see, it works fine once and then comes up with a message to say there is no more DATA. So add line 1110 RESTORE. This will tell the computer to go back and use the DATA again. If you want to, you can put RESTORE 1400, which tells the computer to use the DATA which comes in line 1400 and afterwards. This makes it possible for you to choose one part of the DATA and use it several times over if you want to.

Don't be confused by the fact that string DATA does not need inverted commas. The fact that you must tell the computer to READ A$ or X$ tells it to look for strings—the dollar sign is the key here. It would be very tedious indeed if you had a long list of words to include as DATA and you had to put inverted commas around them all—think of it as a lucky break.

The same applies to INPUT statements—you can type 'yes' or 'no' in response to a question put by an INPUT PROMPT, but you do not need to type inverted commas. The computer is told by the '$' at the end of the variable name to accept a string.

However, you do have to put inverted commas around a DATA or INPUT item if you want to include a comma in the string—otherwise the computer thinks that the comma marks the end of the string.

A function is a kind of 'program within a program', designed to carry out some specific task—a sequence of instructions which is set aside for use whenever you need it, and can be utilized again and again.

As a simple example, suppose that you want a particular message to be displayed on the screen at various different stages of a program. You could define this action as a function, by typing something like:

```
100     DEF WARNING
110         CLEAR SCREEN
120         PRINT AT 10,7: "NOW PAY CLOSE
            ATTENTION..."
130         !
140         SOUND !
150         !
160         !
170     END DEF
```

Line 140 adds a sound signal.

You always have to give the function a name (in this case WARNING), and introduce it by the keyword DEF. The definition which then follows may be one line long or 1000 lines long, but whatever its length it must have the words END DEF as the last line of all.

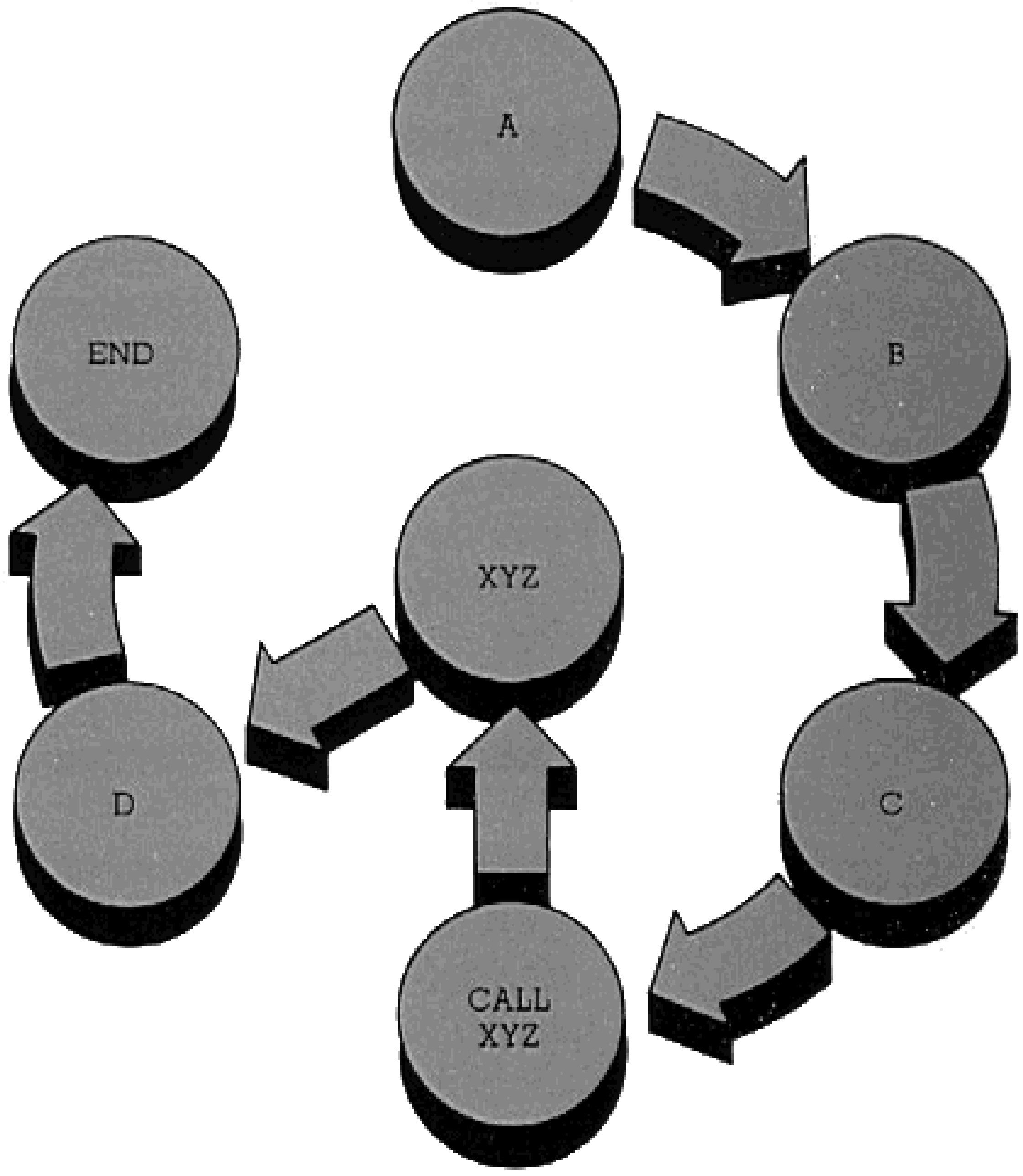
CALLING FUNCTIONS

The function cannot work on its own. Type in the above lines, and try to run them; for the moment, nothing will happen. The function needs to be activated by the statement CALL WARNING. Type this as line 180, and run the program; then type the same instruction in immediate mode. Note that the definition of the function may come after the CALL statement—or indeed after the END statement of the program. If, instead of line 180, you typed:

```
80      CALL WARNING
90      END
```

—the function would still work.

Whenever the computer reaches a CALL statement, it stops whatever it is doing, finds the function being CALLed, goes and carries it out, then returns to the point in the program immediately after the CALL. The diagram should make this clear.



Remember that a function is inactive as long as you don't actually tell the computer to perform it. If the computer is simply following through the sequence of line-numbers and it comes to the part of a program where a function is placed, it jumps over it and does whatever comes afterwards. You cannot make a function work without using its name elsewhere in the program.

LOCAL AND GLOBAL VARIABLES

Usually, functions will handle variables. To make them do this correctly, some important rules must be followed.

Try typing this:

```
100     DEF CUBE
110             INPUT PROMPT "Number to be
                cubed: ":Z
120             PRINT Z; " cubed is  ";Z*Z*Z
130     END DEF
140     CALL CUBE
```

—and after running it, add:

```
150       PRINT Z
```

—and run it again. You will then find that although the CUBE function still works, the computer gives an error message when it comes to line 150. Why is this?

The answer is that, in the present case, Z is what is known as a *local variable*. It belongs exclusively to the CUBE function, and the part of the program outside that function knows nothing about it. Since a function is treated by the computer as a separate little program, it may use its own private variables to help it perform its task. But these private variables mean nothing to the rest of the program; once the task has been completed, their values are thrown away. So, at line 150 above, the computer doesn't know what to print.

Next, renumber line 110 as 90—to place it outside the function. You will now find that the program allows you to type in a number for Z, tells you the cube of this number, and then re-prints the number itself. That is, line 150 no longer confuses it.

The reason is that by introducing Z before the function is called, you have made Z into a *global variable*. A 'global' variable is one that is available to the general 'world' of the program.

Now add:

```
125       LET Z=20
```

—and run the program again. What now happens is that the function takes the number out of the 'box' (labelled Z) in the 'main' part of the program, performs a calculation with it and prints the result, then alters the number itself and puts it back in the same box as before. The 'main program' then prints out this new number.

The point to remember is that if a function contains a line which mentions a variable, and this variable hasn't been introduced before the function is called, the function will treat it as a local, or private, variable. If, on the other hand, the variable has been declared already, the function will regard it as 'global'; any new value which the function gives to it will be passed on to the rest of the program.

You will have read about declaring variables in earlier parts of the manual. (If you need a recap on this, look at pages 24 and 75.) It was stated that, although this is not always essential, it is best to declare every variable you use. Obviously, declaring variables is especially important if you are making much use of functions.

A variable, as you know, can be declared by a NUMERIC or STRING statement, or by the word LET (e.g. LET A=Ø). In previous examples, it didn't much matter which of these forms of declaration was used; but their effects must be precisely understood when you are working with functions. Inside a function, a NUMERIC (or STRING) statement always has the effect of creating a local variable. In the program above (after renumbering the original line 11Ø), try adding:

```
11Ø      NUMERIC Z
115      LET Z=3
```

You will then find that the program operates with two quite separate Z's, one inside the function (a 'local' Z) and one outside it (a 'global' one). On the other hand, if you now delete line 11Ø, there will only be one Z. The LET statement in line 115 will not create a new (local) variable, but will alter the global variable that was introduced by line 9Ø.

The program below contains some rather more complex examples of functions. It's a restructured version of an earlier program which appeared in the chapter about decisions. Apart from showing you what functions look like within a program, it will also demonstrate that there are always several ways of putting a program together. Some look nice, some look horrible, some are incomprehensible, some are very efficient and others are quite the opposite. If you look at the program as a whole, you will probably agree that this version of it is much tidier.

As long as you want the computer to print more names on the screen, the program will not end. You will have to type 'N' when asked, to finish it.

```
1ØØ      DO
11Ø            LET A=Ø
12Ø            LET A$=""
13Ø            !  ________________________________
14Ø            !      Lines 11Ø and 12Ø declare two
```

```
150         !    'global' variables, which the func-
160         !    tions will use (and alter) and then
170         !    hand back to the main program.
180         !    ____________________________________
190         CALL MENU
200         FOR X=1 TO 1500
210         NEXT X
220         !    ____________________________________
230         !    Then comes the main program,
240         !    which begins by calling the menu,
250         !    and, after the menu has finished,
260         !    delays for 3 seconds to give you
270         !    time to read the screen. After you
280         !    have made your choice from the
290         !    menu, the main program goes
300         !    through the CASE block, does what
310         !    you have chosen, and then calls the
320         !    ANSWER function which decides
324         !    whether or not the program will run
327         !    again.
330         !    ____________________________________
340         CLEAR SCREEN
350         SELECT CASE A
360         CASE 1
370               PRINT AT 9,18:"ENTERPRISE!"
380         CASE 2
390               INPUT PROMPT "Then please tell
                  me your name. ":NAME$
400               PRINT AT 9,18:NAME$
410         CASE 3
420               INPUT PROMPT "Then please tell
                  me your name. ":NAME$
430               PRINT AT 9,18:NAME$
440               PRINT AT 11,18: "ENTERPRISE!"
450         END SELECT
460         FOR X=1 TO 3000
470         NEXT X
480         CLEAR SCREEN
490         CALL ANSWER
500     LOOP WHILE A$="Y"
510     END
520     !        ____________________________________
530     !        500 concludes the main loop. If A$
540     !        is "Y", the program goes back to
550     !        the beginning. In effect, the
560     !        program does not end until you
```

```
570     !           reply with "N" (or no, or nope, etc.)
575     !           in the ANSWER function.
580     !
590     DEF MENU
600         CLEAR SCREEN
610         PRINT AT 9,18:"Menu"
620         PRINT AT 11,9:"1) Print my name."
630         PRINT AT 12,9: "2) Print your name."
640         PRINT AT 13,9:"3) Print both of our
            names."
650         PRINT AT 16,1: "Please enter the
            number of your choice: "
660         DO
670             INPUT A
680         LOOP WHILE A<1 OR A>3 OR
            A<>INT(A)
690     END DEF
700     DEF ANSWER
710         PRINT
720         PRINT
730         DO
740             INPUT PROMPT "Would you like
                to do that again? ":A$
750             LET A$=UCASE$(A$(1:1))
760         LOOP UNTIL A$="Y" OR A$="N"
770     END DEF
```

If you remove lines 11Ø and 12Ø, the program won't work any more—because the two variables whose declarations you've missed out have become local to the functions which contain them.

So far we have been using functions that are activated with CALL statements and may produce a variety of effects, such as inviting us to type in more data or printing messages on the screen. We shall now look at a rather different class of function—one which simply has the purpose of handing back a single number to the main part of the program.

Several functions of this kind are supplied 'ready-made' by the computer. Take SQR for instance. A program line can contain the statement PRINT SQR(121), or PRINT SQR(P), or LET M=2*SQR (N)+1. The function SQR calculates the square root of the bracketed number or variable, then lets you use this square root as part of an 'expression' or do whatever else you want with it. You are also familiar with the function INT. Such BASIC words give you the means to perform, quickly and easily, calculations that you may need often.

Suppose you were writing a program that made use of several 'factorial' numbers (factorial 4 means 4*3*2*1; factorial 6 is 6*5*4*3*2*1; etc.). There is no ready-made function to calculate factorials. But if you wanted, you could devise one by the methods you have so far learned. You could type:

```
100     DEF FACT
110     !
120     !
130     !
140     !
150     !
160     !
170             FOR Y=F-1 TO 1 STEP -1
180               LET F=F*Y
190             NEXT Y
195     IF F=0 THEN LET F=1
200     END DEF
```

This function will take a global variable, F, from the 'main' program, alter it and hand it back again.

And then, in order to print (say) factorial 13, or use factorial 7 in an 'expression', you could add:

```
210     LET F=13
220     CALL FACT
230     PRINT F
240     LET F=7
250     CALL FACT
260     LET NUMBER=F*1.5+3
270     PRINT NUMBER
```

But this, as you can see, is a good deal more cumbersome than using a BASIC word like SQR, because every time the function FACT is called, the number on which you want it to operate has first to be placed in the variable F. However, the computer offers you ways of overcoming this limitation. Delete all of the above, and type instead:

```
100     DEF FACT(X)
110             FOR Y=X-1 TO 1 STEP -1
120               LET X=X*Y
```

```
130           NEXT Y
140           LET FACT=X
150       END DEF
160       PRINT FACT (13)
170       LET NUMBER=FACT(7)*1.5+3
180       PRINT NUMBER
```

By doing it like this, you are making FACT into a kind of new BASIC word of your own—which can be used in the same ways (and just as conveniently) as you would use INT or SQR.

The two things about the above program which are new to you are lines 100 and 140. The bracketed X in the DEF line tells the function to look for a bracketed number following the word FACT in a main program line, and automatically put that number into its own local variable X. Line 140, in effect, assigns a value to a variable that has the same name as the function itself, and so allows this value to be handed straight back to the line in the 'main' program where the function is mentioned (see lines 160 and 170). In this way, you do without a CALL statement.

DUMMY VARIABLES

In technical terms, the bracketed X in line 100 above is known as a *dummy variable*. It tells the function to expect one number to be handed to it for processing. (The final example in this chapter will show you a function with two dummy variables, telling it to expect two numbers.) That number may, however, be supplied by a global variable in the main program. Delete lines 160-180, and substitute:

```
160       LET A=11
170       PRINT FACT (A)
```

What happens now is that the function looks into box A and makes a note (takes a copy) of the number it sees there. It then puts an identical number into its own box X, which it uses for its calculations—during which the number in box X changes, but the one in box A stays the same.

The particular way that the dummy variable works is seen if you alter lines 160 and 170 to:

```
160       LET X=11
```

```
170       PRINT FACT (X)
```

You have now defined a global variable with the same name as the function's dummy variable. But you will find that the program still treats these two 'boxes' as separate, even though at the beginning of the function a number is 'copied' from one box to the other. You could add an extra line to the function:

```
145       LET X = 100
```

—and an extra line to the 'main' program:

```
180       PRINT X
```

—but you would find that line 180 printed 11, not 100. Line 145 alters the 'local' X only.

PARAMETER REFERENCING

You have just seen a function, using a dummy variable, perform a calculation with a number 'copied' from a global variable. But although the function handed back a number to the main program, the actual variable from which the copy was taken remained unchanged.

It is a different matter if you put REF (for 'reference') in front of the dummy variable—as the following simple example will show.

This program allows you to type in two numbers—for A and B—and then it changes them by raising A to the power of B, and B to the power of A:

```
100       INPUT PROMPT "First Number":A
110       INPUT PROMPT "Second Number":B
120       CALL POWERS (A,B)
130       PRINT A,B
140       END
150       DEF POWERS (REF X, REF Y)
160              LET Z = X
170              LET X = X^Y
180              LET Y = Y^Z
190       END DEF
```

Line 150 introduces two dummy variables. When the function is CALLed, the value of the first bracketed variable in line 120 is transferred to variable X, and the value of the second one is transferred to Y. This is the same kind of thing that you have seen before, except that in earlier examples there was only one dummy variable; also, since this function will hand back two numbers (not just one) to the main program, it has to be activated with a CALL statement.

The difference made by introducing REF in line 15Ø is simply that when the function has finished its calculations, the new value it has given to X is transferred back into the global variable A (and Y is transferred back to B).

The to-ing and fro-ing of values between functions and other parts of a program is called parameter passing. If the function has the effect of altering those global variables (or arrays, etc.) from which it took its numbers for processing, we call it *parameter referencing*. In the last example, A and B are reference parameters.

The Enterprise's powerful graphics can be used to provide some impressive pictures and visual effects. You may already have realized this from using the demonstration cassette, and some of the programs provided in the earlier parts of the manual show a glimpse of the possibilities of high-resolution graphics.

In the first part of the manual, PRINT AT was explained to you. This command made use of a system which divided the screen up into a number of 'positions', or imaginary squares, so that you could specify where you wanted something to be printed. PRINT AT 1,1 would put a string (or a number) in the top left-hand corner of the display.

The graphics commands use a similar system to put lines and dots on the screen and enable you to make diagrams and pictures. In this case, though, the 'screen positions' are much smaller. Here's a short program which will draw a line.

```
100     GRAPHICS
110     PLOT 640, 360; 1000, 700
120     END
```

You will have come across the first statement in earlier examples. The word GRAPHICS is a quick and simple way of selecting a blank 'page' on which you can make pictures.

The command PLOT is used for making dots or drawing lines. The four numbers that follow PLOT in line 110 correspond to two positions on the graphic page; this program draws a line from the centre of the screen (640, 360) to a position (1000, 700) in the direction of the top right-hand corner.

Notice how much larger these 'co-ordinate' numbers are than the row and column numbers used with PRINT AT. This doesn't mean that they are referring to a bigger area; the point is simply that the 'graphic page' is divided into a much larger number of very small positions, allowing you to place things far more precisely than you could on a 'text' page. That is, the *resolution* when you are dealing with graphics is much higher.

A further difference between 'graphics' and 'text' is that, on the graphic page, the 'origin' (the position 0,0) is in the bottom left-hand corner, and the first number given in a co-ordinate pair is the *horizontal* position. This follows the (x,y) conventions normally used for graph drawing.

You will have noticed, when you ran the program, that four lines from the normal text page were left at the bottom of the display. This allows you to continue typing into the computer while keeping the drawing visible.

This split between a standard graphics page and four lines from the text page is provided to keep operation easy, and always results when you give the simple command GRAPHICS. It leaves you with an area of screen for plotting that measures 1280 positions horizontally by 720 vertically, so that the co-ordinates of the top right-hand corner position are (1279,719).

The two parts of the display can be cleared separately by the commands CLEAR GRAPHICS and CLEAR TEXT, or both together by CLEAR SCREEN.

If you type DISPLAY TEXT, the screen will revert to a full-size text 'page'. Similarly, DISPLAY GRAPHICS switches back to the graphics page without altering anything that was on it before. Notice the difference between these commands and the simple words TEXT and GRAPHICS—which have the effect of *clearing* the text and graphics pages.

When you learn about 'channels' and the more sophisticated features of the graphics, you will be free to specify the size of your 'pages' and display them in any part of the screen you choose.

DOTS OR LINES

Try changing the PLOT statement in the program so that it simply reads:

```
110      PLOT 100, 100
```

—and run the program again. A dot appears on the screen.

Now add a semicolon after 100,100. Then add:

```
115      PLOT 1000,700
```

—and run the program once more. it draws a line again. Then remove the semicolon from line 110. Run the program. Two dots appear. Why?

The answer is that the semicolon controls whether the video 'beam' is left on. When the beam is 'on', it leaves a visible line as it plots between two points. To keep it on, the semicolon is necessary after a PLOT statement.

You can think of the beam as a drawing pen. A PLOT command, with co-ordinates, will put the pen on the paper and plot a dot at least. To draw a line between two screen positions, type the two pairs of co-ordinate numbers and separate them with a semicolon. This changes the command to read: 'PLOT a dot at position (100, 100) and then keep the pen on the paper, moving in a straight line to (1000,700)'. If you left out the semicolon, our imaginary pen would still move, but it wouldn't touch the paper.

You can also use the commands SET BEAM ON and SET BEAM OFF to put the 'pen' on the paper or lift it up.

Here's a measles program:

```
100     RANDOMIZE
110     INPUT PROMPT "How many measles? ": B
120     GRAPHICS
130     LET Z=0
140     DO
150         LET X=RND (1279)
160         LET Y=RND (719)
170         PLOT X,Y
180         LET Z=Z+1
190     LOOP UNTIL Z=B
200     END
```

That program will plot dots in random positions on the screen. By changing 170 to:

```
170 PLOT X,Y;
```

—you could change the measles to lines.

Any of the graphics commands can, of course, be included in the definition of a function. The following example shows, incidentally, that you can put several screen positions in one PLOT command and draw lines between them all:

```
100     DEF DIAGRAM
110         GRAPHICS
120         PLOT 504,544;564,464;516,448;504,544;
            460,464;516,448
```

```
130      END DEF
```

Run this, then type CALL DIAGRAM in immediate mode.

If you put a *comma* after a pair of co-ordinates, no dot will be inserted in that position; the computer will merely move the beam there, and turn it off. You will see that this is necessary if you want to give the instruction PLOT PAINT.

TURTLE COMMANDS

We shall now look at another set of commands which enable you to draw lines. They are called 'turtle' commands, because they were first used for controlling a slow-moving robot animal. This time, we don't need to work out the co-ordinates of a whole series of screen positions.

```
100      OPTION ANGLE DEGREES
110      GRAPHICS
120      PLOT 300,150;
130      PLOT ANGLE 80;
140      PLOT FORWARD 500;
150      PLOT BACK 320;
160      PLOT RIGHT 35;
170      PLOT FORWARD 420;
180      PLOT BACK 285;
190      PLOT RIGHT 100;
200      PLOT FORWARD 340
210      END
```

You can see, sure enough, that this is rather like guiding an animal around the screen. But some of the program lines need a little explanation.

Line 100 tells the computer that we want to measure angles in degrees, not radians. (A radian is approximately 57 degrees; there are some mathematical operations for which radians would be more convenient.)

The GRAPHICS statement (line 110) has the effect of setting the beam to (0,0) and turning it off. (CLEAR GRAPHICS would also do this.) So line 120 is needed, to give the starting position from which we want the beam (our 'animal') to move.

Next, we have to point the animal in the right direction. PLOT ANGLE 0 would leave it facing horizontally towards the right of the screen. PLOT

ANGLE 90 would point it straight upwards. The PLOT ANGLE command says: 'first consider the beam to be facing due right, then turn it anticlockwise through the number of degrees specified'.

A command to PLOT FORWARD or PLOT BACK is followed by the required number of graphic screen positions. PLOT RIGHT or PLOT LEFT makes the animal change course, and is followed by the number of degrees through which we want the beam to be turned, relative to its previous direction. You have seen this before, in the program on page 66.

Notice that with 'turtle' commands you still have to use semicolons to keep the beam switched on. (Try removing some of the semicolons from the program or replacing them with commas, to see what happens.)

ELLIPSES AND CIRCLES

The following program plots an ellipse:

```
100     GRAPHICS
110     PLOT 640,250,
120     PLOT ELLIPSE 100,200,
130     END
```

Line 110 gives the centre of the ellipse. The first number after PLOT ELLIPSE is the horizontal distance (in screen positions) between the centre and the circumference, and the next number is the vertical distance. If these two numbers were the same, the program would draw a circle. Notice the commas at the ends of lines 110 and 120. If you missed out either of these, the centre of the ellipse would be marked on the screen by a dot. As it is, the program leaves the beam in this centre position but turns it off.

COLOURS

You are probably well aware that the Enterprise can display 256 colours. Up until now, you have not had many chances to use them. This is where you learn to master the many hues the Enterprise can put on its screen.

The following program will display all 256 colours at once:

```
100     !
110     GRAPHICS 256 !   Note the number which this
120     !                time has to follow
```

```
125     !                    GRAPHICS.
130     !
140     LET Z=0
150     FOR Y=0 TO 560 STEP 80
160        FOR X=32 TO 1052 STEP 32
170           SET INK Z
180           PLOT X, Y; X, Y+70
190           LET Z=Z+1
200        NEXT X
210     NEXT Y
220     END
```

The Enterprise identifies each colour by a code-number in the range 0-255. For simplicity, a special function 'RGB' is supplied to allow you to select colours by mixing amounts of red, green and blue. This is explained later in the chapter.

COLOUR MODES

In the above program, you had to type the number 256 after the word GRAPHICS, to tell the computer to make all its colours available to you at once. That is, you had to select the appropriate *colour-mode*.

Notice that in this program, the lines drawn on the screen are thicker than those you have seen in previous graphics programs. The point is that the more colours you have at your disposal, the less fine your drawings can be. This is necessary to conserve memory space in the computer.

We have said that the 'graphics' page gives a higher 'resolution' (degree of precision) than a 'text' page. But we must now consider the further differences of resolution that depend on which colour-mode is being used. There are four such high-resolution (HIRES) modes, and each gives a particular trade-off between the number of colours you can display and the number of 'dots' per horizontal row that are made available for plotting. Here they are:

GRAPHICS HIRES 2—When this command has been given, only 2 colours can be displayed at a time, but you have 640 separate dots across the width of the screen.

GRAPHICS HIRES 4—With four available colours, you have 320 dots per horizontal line.

GRAPHICS HIRES 16—This time, there are 160 dots per line.

GRAPHICS HIRES 256—With the possibility of displaying as many as 256 colours at once, you are given 8Ø separate dots per line.

If you type simply GRAPHICS, the computer will use the same mode as it did last time you gave the command. When typed for the first time after the computer is switched on or reset, GRAPHICS is equivalent to GRAPHICS HIRES 4.

The number of dots vertically is not affected by the colour mode; on the standard graphics page (with the 4 lines of text at the bottom), it is always 18Ø.

Despite the difference in resolution, all the colour modes use the same system of co-ordinates. In other words, PLOT Ø,Ø;64Ø,36Ø will always draw a line from the bottom left-hand corner to the middle of the 'page', although the fineness of the drawing will vary with the mode. (The same co-ordinate scheme could actually be used for a resolution twice as high as is possible even on the Enterprise.)

The Reference Section explains how the resolution can be halved (and memory saved) by giving the command GRAPHICS LORES. It also gives details of the so-called 'attribute' mode of graphics.

GRAPHICS MODES

In addition to HIRES, there are two other graphics modes.

LORES is identical to HIRES, but uses less computer memory and provides half the horizontal resolution. Colour modes are specified in the same manner as HIRES, eg:

```
GRAPHICS LORES 16
```

would give a 16-colour low resolution graphics page.

ATTRIBUTE mode is a special form of video display, which is a cross between text and graphics modes. It can be used for character printing, or for plotting commands, and provides 16 colours, but needs careful handling of the ATTRIBUTES 'flag' for most effective use. See the reference section, page 188.

No colour mode should be specified, the command is:

```
GRAPHICS ATTRIBUTE
```

SELECTING COLOURS

Let's turn again to the 256-colour mode. In this mode you can draw shapes in whatever colour you like, by preceding the plotting commands with the statement SET INK and the code-number of the colour. Also, before using CLEAR GRAPHICS, you can type SET

PAPER..., so as to choose the colour of the 'background'.

Experience could teach you which colour corresponds to which number: 18 is a light green, 91 is bright yellow, etc. But if you want to use a particular colour and don't happen to know what code-number goes with it, there is an alternative way to specify it.

Every possible colour can be created by a combination of red, green and blue. For example, white comes from mixing all three of these colours. Yellow is produced by mixing just red and green. (This may surprise you, but bear in mind that mixing the actual sources of light gives rather different results from mixing paint.) Black is no colour at all. Complex colours are created by mixing red, green and blue in varying amounts.

This is the principle by which the colours on the Enterprise work; you can define any colour as a mixture, by typing the word RGB, followed (in brackets) by three numbers with commas separating them. These numbers, which must be in the range Ø-1, define the proportions of red, green and blue (respectively) that you want to be mixed.

So SET INK RGB (1,.5,.5) would give you pink as a plotting colour. RGB (.4,.4,Ø) is a dull yellow; RGB (.6,.6,.4) is a shade of grey; and so on.

The 8 colours below can be selected very simply, by just typing their names (e.g. SET INK GREEN). Here are the 'mixtures' to which they correspond:—

BLACK	=	RGB (Ø,Ø,Ø)
RED	=	RGB (1,Ø,Ø)
GREEN	=	RGB (Ø,1,Ø)
YELLOW	=	RGB (1,1,Ø)
BLUE	=	RGB (Ø,Ø,1)
MAGENTA	=	RGB (1,Ø,1)
CYAN	=	RGB (Ø,1,1)
WHITE	=	RGB (1,1,1)

THE PALETTE

If you give the command GRAPHICS HIRES 16, restricting yourself to 16 colours on the display at any one time, you still have considerable freedom to choose which ones they will be. You do this by specifying a 'palette'—a list of selected colours which are to be made available for drawing.

First, type SET PALETTE, then list eight of the colours that you want to use. These can be freely chosen from the full range of 256, and you can specify them by their standard code-numbers, by their names (if they are in the above list), or by defining them as a 'mixture'. For example, you could type:

```
100     SET PALETTE 67,31,WHITE,
        4,RGB(Ø.,3,.8),RGB(.7,.7,.1),
        187,19Ø
```

—and these colours would then be numbered Ø-7 in your 'palette'.

For your remaining eight colours, you have less freedom of choice. The colours numbered 8-15 in your palette all have to belong to a single group of related colours. You select them with the command SET BIAS, followed by any number belonging to the group that you want. For example, if you typed SET BIAS 67 (or any other number in the range 64-71), then the colour with the standard code-number 64 (the first in this group) would become number 8 in your palette; standard number 65 would become palette-number 9; and so on. You can imagine that, with SET BIAS, you are specifying the 'filter' or the 'wash' to be laid over the 'Telextext primary' colours.

You can now select any colour in the palette to make the 'ink' for plotting the next line or shape. An important point is that, if you are not using the 256-colour mode, any command such as SET INK 6 or SET PAPER 6 will refer to the colour listed as number 6 within the palette, not the one with 6 as its standard code-number. Also, don't forget that the numbers in the palette count upwards from Ø, not from 1.

In the 4-colour mode, only the colours numbered Ø-3 in the palette can be used, so there is no point in listing more than four colours in a SET PALETTE command. Similarly, in the 2-colour mode, you will only want to specify palette colours Ø and 1.

Sometimes you may want to alter just one colour in the palette, while leaving the rest as they are. This example alters palette colour 3:

```
SET COLOUR 3, 11Ø
```

You can also, of course, use any of the modes without actually bothering to select your palette. If nothing is specified by you, the computer will always use certain pre-programmed 'default' colours.

USING THE PALETTE

Remember that if you alter the set of colours in your palette, this will affect not only the lines and shapes that are to be drawn afterwards, but also those already on the screen. This ability to change all colours on a display with only a single command forms the basis of some of the most dramatic and fast-moving graphic effects.

In the following program, which creates ellipses of random sizes and colours, we shall start by making all the palette colours the same; so a line drawn in 'ink' of colour number 2 will look no different from one drawn in colour 3, and both will be distinguishable from the 'paper'—the drawing will, in fact, be invisible. Then, when a key is pressed, the drawing will stop, and by repeatedly varying the contents of the palette, the program will make the different 'inks' change colour and stand out against each other and the background.

The program ends with an infinite loop—it will never finish unless interrupted. To halt execution, press the 'stop' key.

```
100    RANDOMIZE
140    GRAPHICS HIRES 4
150    SET PALETTE BLUE,BLUE,BLUE,BLUE
160    !
170    !                  Invisible display.
180    !
190    DO
200         SET INK RND*3+1
210         PLOT 625,330,
220         PLOT ELLIPSE RND*500,RND*300,
230    LOOP WHILE INKEY$=""
240    !
250    !       When key is pressed, show display.
260    !
270    DO
280         SET PALETTE BLUE,BLUE,RED,GREEN
290         !
300         FOR X=1 TO 500 !       Delay for
310         NEXT X !               nearly 1 second.
320         !
330         SET PALETTE BLUE,RED,GREEN,BLUE
340         FOR X=1 TO 500
```

```
350           NEXT X
360           SET PALETTE BLUE,GREEN,BLUE,RED
370           FOR X=1 TO 500
380           NEXT X
390       LOOP
400       !  ----------------------------------------
410       !      Use 'stop' key to halt program.
420       !  ----------------------------------------
```

Note, by the way, that the PALETTE, INK and PAPER commands can also be used if you are working with a 'text' page. The 'Video Options' chapter in the Reference Section gives details of this. Try the following experiments:
SET £102: COLOUR 1, MAGENTA
SET £102: INK 3

PLOT PAINT

This instruction fills a solid shape with the current 'ink' colour. Here is a program which will draw one circle inside another and then paint in two different colours:

```
100       GRAPHICS HIRES 4
110       SET PALETTE WHITE, YELLOW, BLUE
120       PLOT 400,400,
130       PLOT ELLIPSE 200,200,
140       PLOT ELLIPSE 80,80,
150       SET INK 3
160       PLOT PAINT
170       PLOT 400,250,
180       SET INK 2
190       PLOT PAINT
200       END
```

So the area filled in by PLOT PAINT is one which currently contains the 'beam' and is enclosed by a continuous line of a different colour from the beam position. Any gaps in the line will mean that the painting will go outside the shape and try to fill the whole screen. Notice the commas at the ends of program lines 130, 140 and 170; we had to prevent a dot from appearing in the beam position, since otherwise PLOT PAINT would have painted this dot only.

LINE STYLE AND LINE MODE

LINE STYLE allows you to plot with various kinds of broken line. For example, if you type SET LINE STYLE

1Ø, all the lines drawn (until the style is re-set) will be made up of long, closely-spaced dashes. SET LINE STYLE 9 gives you lines of alternate dashes and dots. There are 14 line-styles (numbered upwards from 1) for you to experiment with.

With the command SET LINE MODE, you can determine how lines plotted on the screen will interact with shapes that are there already. In line mode Ø (the 'default'), any new line will simply 'overwrite' lines or shapes previously drawn. In other modes (numbered 1-3), the old and new 'inks' on the page will combine in various ways to determine the plotting colour; the Reference Section gives further details.

PAGES AND CHANNELS

So far, you have been using the command GRAPHICS (or TEXT) to create a blank 'page' onto which you can draw (or write). Most of the time you will probably find this arrangement adequate, but as you become more experienced you may wish to make use of the greater flexibility that comes from specifying 'channel' numbers (normally in the range 1-1ØØ) for your various text and graphics 'pages'. The following should be read in conjunction with the chapter on 'Channels' and the relevant parts of the Reference Section (see e.g. 'Video Options' and the keyword OPEN).

There are two main advantages in opening new 'channels' for your pages:
(1) Several different pages, containing complex drawings or text, can be kept in the computer's memory at once; any of them can then be displayed by a single command.
(2) You can specify the size of a page. So you can make a graphics drawing fill (almost) the whole screen (but note that you cannot use the 'status line' at the top); or you can save memory by making the page small. You can choose the vertical position on the screen for displaying the page or a part of it.

The following example creates a small text page and shows it in the middle of the screen:

```
 5Ø     SET BORDER CYAN
1ØØ     SET VIDEO MODE Ø
11Ø     SET VIDEO COLOUR Ø
12Ø     SET VIDEO X 2Ø
13Ø     SET VIDEO Y 1Ø
14Ø     OPEN £1: "VIDEO:"
```

```
150      DISPLAY £1:AT 7 FROM 1 TO 10
160      PRINT £1: "A small text page..."
170      END
```

Line 140 assigns channel number 1 to our new video page. But the 'video mode', 'colour mode', and page dimensions have to be specified before this is done.

Video mode 0 is a 40-column text page; mode 1 is a graphics page; mode 2 is 80-column text.

VIDEO COLOUR selects the colour-mode, according to the following convention:

VIDEO COLOUR 0	—	2-colour mode
VIDEO COLOUR 1	—	4-colour mode
VIDEO COLOUR 2	—	16-colour mode
VIDEO COLOUR 3	—	256-colour mode

A text page should always be in colour mode 0; this still allows you some choice of colours.

Lines 120 and 130 give the width and height of the page, in 'character positions'.

Line 150 is an instruction to put the top part of the page (measured as 10 character-rows) onto the display, starting from screen row 7. In this case, of course, it displays the page as a whole.

Notice that the PRINT command in line 160 has to include the channel number.

The height specified for the page can, if you like, be greater than the height of the screen. (The maximum is 255 character-rows.) The following program defines a page measuring 8 columns across by 30 rows down, and plots an ellipse on it; then two segments, containing the bottom and top portions of the drawing, are displayed in turn.

```
100      SET VIDEO MODE 1
110      SET VIDEO COLOUR 2
120      SET VIDEO X 8
130      SET VIDEO Y 30
140      OPEN £1: "VIDEO:"
150      PLOT £1:128,540,
160      PLOT £1:ELLIPSE 115,500,
170      DISPLAY £1:AT 10 FROM 21 TO 30
180      FOR X=1 TO 1500
190      NEXT X
200      DISPLAY £1:AT 10 FROM 1 TO 10
```

```
210     END
```

The next example defines 3 'pages' and displays them simultaneously in different parts of the screen:

```
100     SET VIDEO MODE Ø
110     SET VIDEO COLOUR Ø
120     SET VIDEO X 42
130     SET VIDEO Y 8
140     !
150     OPEN £1: "VIDEO:" !                      Text page.
160     !
170     SET VIDEO MODE 1
180     SET VIDEO COLOUR 3
190     !
200     OPEN £2: "VIDEO:" !        256-colour graphics.
210     !
220     SET VIDEO COLOUR 1
230     !
240     OPEN £3: "VIDEO:" !          4-colour graphics.
250     !
260     DISPLAY £1: AT 9 FROM 1 TO 8
270     DISPLAY £2: AT 1 FROM 1 TO 8
280     DISPLAY £3: AT 17 FROM 1 TO 8
290     PRINT £1: "Text..."
300     SET £2: BEAM ON
310     PLOT £2: 1ØØ,1ØØ;
320     SET £3: BEAM ON
330     PLOT £3: 1ØØ,1ØØ;
340     END
```

After running this, you won't be able to see what your are typing. Type TEXT or press function key 5 to return to normal.

Note that every 'page' has its own 'palette', although a SET BIAS command will be applied to all pages.

The use of channel numbers enables you to draw or write on a page even if it isn't currently displayed. It also allows characters to be printed on a grahics page, at the current 'beam' position. For example, if channel 3 has been opened as a graphics page, you can type something like:

```
PLOT £3: 64Ø, 5Ø
PRINT £3: "Hello"
```

—and the string will be added to the page, whether or not it is at present on view.

SET BORDER

As used in the first example of this section, you can select a colour for the border round the whole visible display—the 'desk' onto which the various text or graphics 'pages' are placed. The SET BORDER command is independent of all 'palette' commands (which only apply to particular pages), so this colour must be set by specifying a standard code-number, a colour name, or a 'mixture'. For example, SET BORDER 255, SET BORDER WHITE or SET BORDER RGB (1,1,1) will give a white border to the whole display.

If channel 1Ø1 (the channel of the standard graphics page) is not open when the border colour is set, the command must be given with a suitable channel number (usually £1Ø2, the number for the standard 'text' page)—e.g. SET £1Ø2: BORDER 116.

The Enterprise is provided with a pre-defined group of characters which make up the standard character set. This follows the International Standards Organisation (ISO) character definitions, but is commonly referred to as ASCII—which stands for American Standard Code for Information Interchange.

Each character in the character set has a code-number in the range between 32 and 159. You can refer to a standard character by its appearance, as in PRINT "N", or you can refer to it by its code. PRINT CHR$(78) is the same as PRINT "N"—try them both.

The computer uses these codes to refer to characters inside itself, but you won't need to see how it does this until you become really advanced. As a simple explanation: if, for example, you press a key on the keyboard—say it was 'a'—the computer will receive a signal to tell it that this key has been pressed, register instantly the code for the character (not the shape), send that code to the part of the computer which controls the screen and translates the code into the details for displaying the character—which will then appear on the display.

As far as you are concerned, all that happens is that you press a key and the character appears at the same time. This is an example of how fast a computer works, especially when you consider that the operation described above is actually far more complex; each phase of that task is broken down into far smaller operations.

Here is a program which will print out the whole character set and then allow you to type in a character for which the ASCII code will be printed:

```
100    FOR N=33 TO 159
110    !
125    !    There are 128 pre-set characters.
130    !    This FOR/NEXT loop allows you to
140    !    miss out the 'control' characters,
150    !    which are Ø to 32 (32 is 'space').
160    !
170         PRINT CHR$(N),
180    NEXT N
190    DO
195         DO
200              INPUT PROMPT "Type a character
                 and its code will appear: ":C$
```

```
210             IF LEN(C$) > 1 THEN
220               PRINT "Only one character at a
                  time, please!"
221               !
222               !    The small IF block (210-235)
223               !    makes sure you do not type in
224               !    more than one character. Notice
225               !    that throughout this program, DO
226               !    LOOPS control it. LEN gives the
227               !    length of a string (line 210).
228               !
235             END IF
240           LOOP WHILE LEN(C$) > 1
250           PRINT"The ASCII code for ";C$
260           PRINT "is:";ORD(C$)
261           !
262           ! ORD gives the ASCII code for a
263           ! character.
264           !
265           DO
270             INPUT PROMPT "More characters,
                y/n? ":A$
280           LOOP WHILE A$ < > "y" AND A$ < >
              "n"
290       LOOP WHILE A$="y"
300       END
```

As you can see, ORD and CHR$ are opposites. ORD gives an ASCII code for a character and CHR$ will print a character for an ASCII code.

DESIGNING YOUR OWN CHARACTERS

Now you've read about ASCII codes, a few words on making up your own character shapes are appropriate. For each ASCII code, the computer recalls a shape from its memory—but you can re-define this shape.

Imagine a character is made up of nine rows of eight little lanterns. To form a character shape, some lanterns will be lit and others will not. This forms a pattern of tiny dots which are so small and so close together they appear to join up. So to design or redesign a character, it's helpful to draw a square grid of 8×9 spaces. Then you can form a character by putting dots in the appropriate squares.

Now, to program this information into your computer you need to imagine all the dots are 1s and all the blanks are Øs. By looking at each row you can put together a sequence of Øs and 1s. The program below designs a little character and then prints it.

```
100     NUMERIC N (1 TO 9)
110     !
120     !     A standard character is 8 dots wide and
130     !     9 rows deep. So each row is defined by
140     !     an eight-digit number. A series of nine
150     !     such numbers makes up the definition of
160     !     the character. Lines 28Ø-3ØØ store these
170     !     numbers in the array N.
180     !
190     DATA ØØ11111Ø
200     DATA Ø1ØØØØØ1
210     DATA Ø1Ø1Ø1Ø1
220     DATA Ø1ØØØØØ1
230     DATA ØØ1ØØØ1Ø
240     DATA ØØØ1Ø1ØØ
250     DATA ØØØØ1ØØØ
260     DATA ØØØØØØØØ
270     DATA ØØØØØØØØ
280     FOR ROW = 1 TO 9
290           READ N (ROW)
300     NEXT ROW
310     SET CHARACTER 63,BIN (N(1)), BIN(N(2)),
        BIN(N(3)), BIN (N(4)), BIN(N(5)), BIN(N(6)),
        BIN(N(7)), BIN(N(8)), BIN(N(9))
320     !
330     !     BIN tells the computer to treat the 1's
340     !     and Ø's as binary digits.
350     !
360     PRINT "?"
370     PRINT CHR$ (63)
380     END
```

Line 31Ø is the statement which stores the information for your character. What you have done is re-defined the question mark! The number 63 is the ASCII code, and all those Ø's and 1's are the lights and blanks (or dots and spaces) from which the character is formed. The character is then printed by the command PRINT "?" or PRINT CHR$(63). Also, try typing the question mark in immediate mode.

Once you have run a program which redefines characters, you can still print those characters until you type CLEAR FONT, press the 'reset' button twice in quick succession, or switch off the computer.

It's easy to see how much fun you can have with these user-defined characters, as they are known. You could redesign the entire alphabet, for instance. Programming in gothic or italic script could introduce a whole new dimension to your computing activities! Space games will be all the more vivid for a few custom-made aliens (you could make one creature out of several characters, by printing them all together). The demonstration cassette contains a program which gives you several characters ready-made and another one which allows you to do your own and save them on the cassette.

Sound effects are a valuable asset to most programs, especially games. 'Serious' programs can use noises as signals. Games are much more absorbing for the addition of a few timely and appropriate sounds.

The Enterprise offers you the possibility of listening to the sounds in stereo—on headphones or through your hi-fi. If you want to do this, and you don't like the built-in mono loudspeaker to be heard at the same time, you can silence it by typing SET SPEAKER OFF or pressing 'shift' with function key 7.

Control of the sounds is provided by two BASIC keywords. These are SOUND and ENVELOPE. A SOUND statement gives the computer general information about how long a sound is to last, what pitch it will begin from, what its maximum volume will be, and certain other points. An ENVELOPE statement is a set of instructions specifying in detail the variations of pitch and volume that the sound will undergo during the time it is being played.

THE SOUND STATEMENT

The following is an example of a SOUND statement:

```
100     SOUND PITCH 40, LEFT 127, RIGHT 191,
        DURATION 200, ENVELOPE 10
```

Let's consider one by one the various items (divided from each other by commas) that the program line specifies.

First, the number after PITCH states how high the note will be when it begins to be played. This number can, in theory, be anything from 0 (which actually would make the sound almost inaudibly low) to 127. The range in which good results are normally obtained goes up as far as about 83. Within this range, each increase of 1 will raise the sound by one semitone. Pitch value 37 is equivalent to 'middle C'. If no pitch value is stated, 37 is used as the 'default'.

The next item in the series determines the volume of sound that will be sent to the left speaker. The number after LEFT must be in the range 0-255. If LEFT 0 is specified, the speaker is silenced. LEFT 255 (which is the 'default' value) permits the sound to rise to the loudest volume that the machine can produce—subject to the further instructions contained in an 'envelope'. In the above example, LEFT 127 means that the sound sent to the left speaker will never rise above half-volume, whatever the envelope may stipulate.

RIGHT similarly gives the maximum volume for the right speaker. In our example, this speaker is set at three-quarters volume.

The number after DURATION measures the length of the sound, in 'ticks'—one tick is one-fiftieth of a second, so in this case the sound will last for 4 seconds. (The default is 5Ø ticks.)

ENVELOPE refers to the number of the ENVELOPE statement which is to be used in conjunction with this SOUND statement. There is one built-in envelope, numbered 255, which is used by default; valid numbers for your own envelopes are in the range Ø-254.

Note that all these items following the keyword may be arranged in any order. There are some more things that can be added to the list; we'll come to them later. Since in all cases there are 'default' values, the word SOUND will, of course, produce an effect if used just by itself.

ENVELOPES

This is an example of an envelope:

```
9Ø      ENVELOPE NUMBER 1Ø;2, 8, 63, 5Ø; Ø, 24,
        -16, 1ØØ; -5, 47, -39, 5Ø
```

Once again, the keyword is followed by a fairly long list of data.

To begin with, NUMBER 1Ø identifies the envelope so that SOUND statements can refer to it (see above).

Then comes a semicolon, followed by a batch of four numbers (separated from each other by commas). These numbers define the changes that the volume and pitch will undergo during the first 'phase'—the first portion of the time which has been allotted for the sound (in this case, the phase lasts one second).

The first number in the batch signifies that during this period the pitch will rise by 2 semitones from its starting value as given by the SOUND statement. Instead, −1.5 would lower the pitch by three quarter-tones; Ø would give a constant pitch; etc.

The next number (8) specifies the change in volume, for the left speaker. In an ENVELOPE statement, a unit of volume is equal to one sixty-fourth of the maximum permitted for the speaker—i.e. one sixty-fourth of the volume laid down in the SOUND statement. The number 63 will always raise the volume to the maximum, while −63 will reduce it to silence (any overshoot is ignored).

At the very beginning of its duration, the sound has of course no volume at all. So, in our example, the volume of sound sent to the left speaker will rise, during the first phase of the envelope, from zero to one-eighth of the maximum.

The third number in the batch (47) has a similar purpose to the second, only it applies to the right speaker.

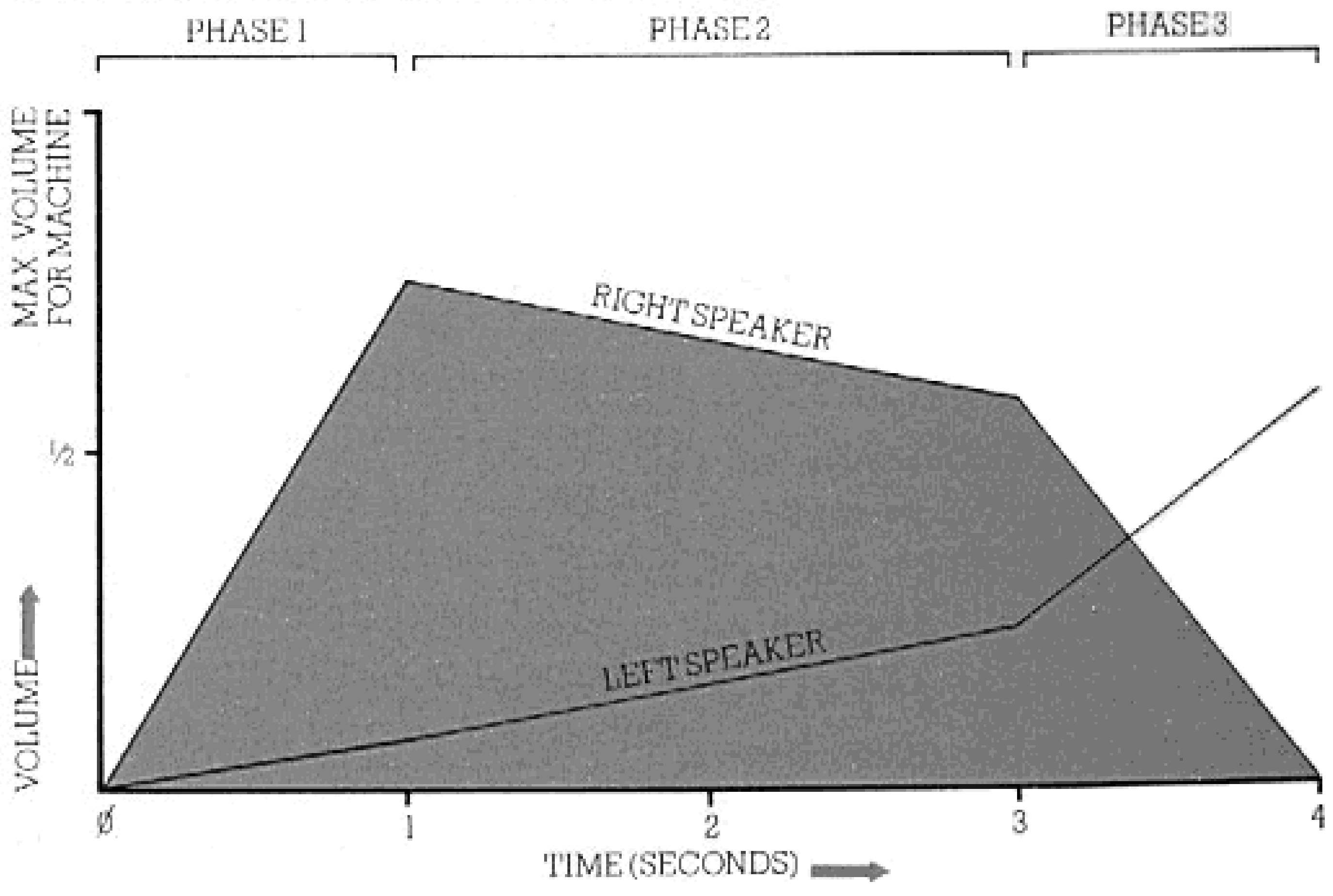
The SOUND and ENVELOPE statements work together, then, to produce two values for the volume-level at any particular moment—one value for the left and one for the right speaker. Note that if you aren't using stereo equipment, the volume at any time will be decided simply by adding together the two separate values.

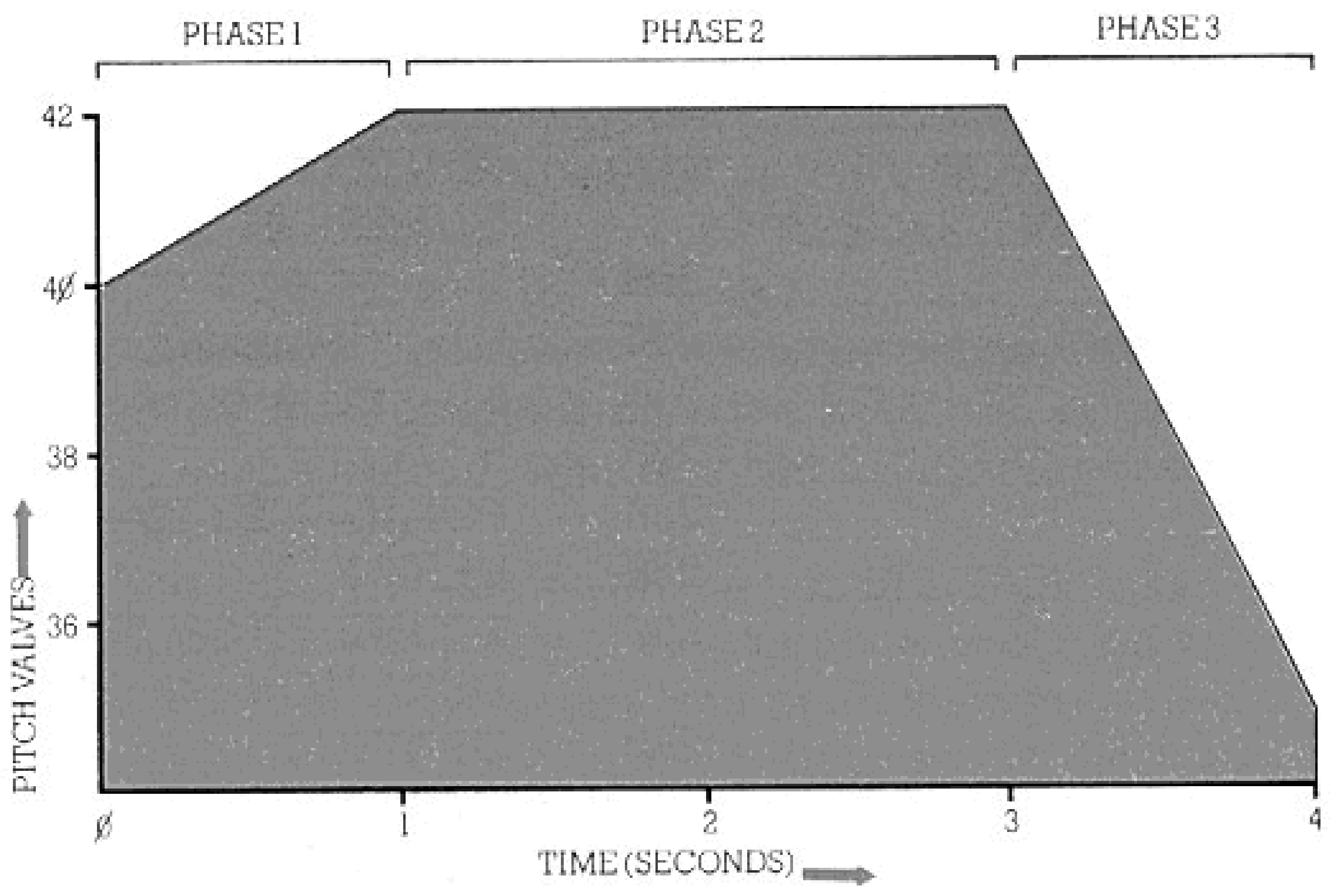
The fourth number in the batch (5Ø) specifies how much time this first phase of the envelope will take. Here again, time is measured in 'ticks'; fifty ticks are equal to one second.

Then we come to another set of four numbers, following another semicolon; these work in the same way as the first batch, except that they define what will happen in the second phase of the envelope.

Then four further numbers (again with a semicolon in front of them) define the third phase.

If you type in and run the two program lines we've been looking at, they will combine to produce a sound that can be represented by this pair of graphs:-





Sounds can be much more complex than this, of course. We shall later see how an envelope can be defined with any number of phases up to 255.

Remember that all instructions contained in an envelope are *relative*—their results depend on the volume range and initial pitch that the SOUND statement specifies. The same envelope may therefore be used with a number of different SOUND statements.

Remember also that the envelope must be defined on a program line that is executed before any sound statement referring to it.

SOUND QUEUES

Playing a sound doesn't hold up any other activity that the computer has been told to perform. In the following program, print and graphics commands are carried out with sounds as an accompaniment:

```
1ØØ     ENVELOPE NUMBER 2Ø; –2, 63, 63, 1ØØ; –3,
        Ø, Ø, 1ØØ; 3, –36, –36, 1ØØ; 2, –12, –12, 1ØØ; Ø,
        Ø, Ø, 1ØØ
11Ø     SOUND PITCH 61, LEFT 255, RIGHT Ø,
        DURATION 5ØØ, ENVELOPE 2Ø
12Ø     CLEAR SCREEN
13Ø     PRINT AT 1Ø,5: "A graph will presently be
        drawn"
14Ø     PRINT AT 11,5: "to represent the envelope
        that"
15Ø     PRINT AT 12,5: "these sounds are using.
        Volume"
16Ø     PRINT AT 13,5: "will be indicated by the blue"
```

```
170     PRINT AT 14,5: "area, pitch by the yellow
        line."
180     SOUND PITCH 43, LEFT 127, RIGHT 127,
        DURATION 200, ENVELOPE 20
190     !
200     !     The sound in line 180 will stop after
210     !     the first two phases of the envelope.
220     !
230     FOR X=1 TO 2000
240     NEXT X
250     SOUND PITCH 25, LEFT 0, RIGHT 255,
        DURATION 500, ENVELOPE 20
260     GRAPHICS
270     PLOT 0, 0; 250, 540; 500, 540; 750, 270; 1000,
        180; 1250, 180; 1250, 0; 0, 0
280     PLOT 100, 20,
290     PLOT PAINT
300     SET INK 3
310     PLOT 0, 450; 250, 350; 500, 100; 750, 350; 1000,
        450; 1250, 450
320     END
```

At line 110, the computer registers your instructions and starts to play the sound; but it doesn't wait for this sound to finish before going on to the next program line. The sound from line 110 carries on while the computer is printing its message on the screen (lines 130-170). At line 180, a further SOUND statement occurs. What happens now is that the computer will register this statement and will aim to play the new sound as soon as the old one finishes—in other words, the sound specified in line 120 is placed in a 'queue' behind the sound from line 110. Similarly, at line 250, the third sound is put at the back of the queue behind the second one. The graph (lines 260-310) is now drawn while one sound follows on after another.

RELEASE

The Enterprise allows one or more phases at the end of a sound envelope to be treated a little differently from the rest. These phases are preceded by the word RELEASE. Conventionally, the 'release' stage of an envelope means a period when the sound is allowed to tail off—when it has effectively finished and only its residual effects are wanted.

Take this example:

```
100     ENVELOPE NUMBER 4;. 1, 63, 40, 5; -. 3,
        -32, -20, 20; 2, 0, 0, 25; RELEASE; 0, -16,
        -10, 10; 0, -15, -10, 15;
```

Here, the three phases before the RELEASE stage take a total of one second, while the two RELEASE phases take half a second.

The difference between the non-RELEASE and RELEASE phases is this: if the 'duration' specified in the SOUND statement runs out before the non-RELEASE phases have finished, they are cut short; but the RELEASE phases are carried out even when the 'duration' is over, provided that no other sound is waiting in the queue.

So if DURATION 25 is specified in a SOUND statement that uses the above envelope, the third phase will not be carried out; the computer will jump straight from the second phase to the RELEASE phases—or to the next sound in the queue, if there is one.

In the case of DURATION 60, on the other hand, the computer will execute the three non-RELEASE phases and the first RELEASE phase—and will then go on to the next sound if there is one waiting.

In the case of DURATION 100, all phases will be carried out, and there will be a pause of half a second before the next sound begins.

INTERRUPT

You have now learnt the essentials of controlling the sounds, but you still have several interesting features to explore. For example, you can make one sound interrupt another. Go back to the program that drew a graph with sound effects in the background. Remove lines 130-170, 230-240 and 260-310, and insert instead:

```
260     ENVELOPE NUMBER 30; 0, 63, 63, 1; 0, 0, 0, 24
270     PRINT "Press i when you want to interrupt this
        sound."
280     DO
290     LOOP UNTIL INKEY$="i"
300     SOUND PITCH 37, LEFT 255, RIGHT 255,
        DURATION 25, ENVELOPE 30, INTERRUPT
```

The inclusion of INTERRUPT in line 300 means that when this line is reached, the sound that is being played will be cut short by the new one—and anything waiting in the queue will be disregarded.

SOUND SOURCES

It is possible to play more than one sound at once. The Enterprise incorporates four sound sources, each of which can have its own independent 'queue' of sounds.

To put a sound into the queue for (say) tone generator 2, all you do is include SOURCE 2 in your SOUND statement. The tone generators are numbered Ø-3. Number 3 is the so-called 'noise generator' (which ignores pitch values). So far, you have only been using tone generator Ø, which is the 'default' source.

When playing various sounds together, you will naturally want to ensure accurate synchronization between the different sound sources. By putting the instruction SYNC into a SOUND statement, you can make one sound begin at precisely the same moment as another—or two or three others, depending on the number inserted after SYNC.

Type in a number of SOUND statements, sharing them between SOURCE Ø, SOURCE 1 and SOURCE 2—i.e. make three separate 'queues'. To the first program line in each queue, add the instruction SYNC 2, so that it will look something like this:

```
12Ø     SOUND PITCH 4Ø, LEFT 255, RIGHT 127,
        DURATION 2ØØ, ENVELOPE 6, SOURCE 1,
        SYNC 2
```

When the lines are run (assuming that you have entered suitable envelopes too), all three sound queues will start up at the same moment.

To be precise, the instruction SYNC 2 says: 'When this sound comes to the head of its queue, hold it up until two more new sounds in other queues are ready to begin.' This means that when one other sound is ready, it too will automatically be held up—until a new sound comes to the front of the third queue, when all three queues will start moving together.

The command CLEAR QUEUE, followed by the number of a tone generator, will silence any sound currently coming from this source and clear away anything waiting in the queue. Note that a SOUND statement with an INTERRUPT instruction will not break off any sounds that are coming from other 'sources'.

MORE COMPLEX SOUNDS

Unless instructed otherwise, the computer assumes that all your sound envelopes will contain between 1 and 20 'phases'. However, if (for example) you want to define envelopes with up to 25 phases, you can type:

```
100     CLOSE £103
110     SET SOUND BUFFER 25
120     OPEN £103: "SOUND:"
```

—and the computer will make available the extra memory space for the purpose. The above lines close and re-open a 'channel'; for an explanation of how channels work, see the separate chapter on the topic, and also the Reference Section (under the keyword OPEN). Channel 103 is the normal 'default' channel for sound output.

The number of phases specified by the SET SOUND BUFFER command can be anything from 1 to 255, although in practice the complexity of an envelope is limited by the maximum length of a BASIC program line (250 characters). SET SOUND BUFFER only affects a channel which is opened subsequently, not one that is open already—which is the reason for lines 100 and 120 above.

Lastly, a SOUND statement may contain the word STYLE followed by a number in the range 0-255. Experimenting with the effects of different sound styles is particularly interesting when you are using the 'noise' generator (sound source 3). For this, see the Reference Section under the heading 'Sound Options'.

MAKING PROGRAMS OUT OF PROBLEMS

We've spent a lot of time dividing programming into different sections and dealing with these one at a time.

We haven't yet concentrated on how a task should be looked at in terms of the way a computer works, or how to put tasks into a program in such a way that you will find it easy to understand long after you have written it. We'll deal with that now.

The problem used here is easy to understand because it's a simple one. Once you have grasped the principle of planning programs, more and more complex problems will seem easier to solve.

THE PROBLEM

We'll deal with the problem of working out how much less it would cost you to buy goods with a percentage reduction on them.

To solve this problem, you need to know only two things: the original price and the percentage by which the shop is offering to reduce it. From these two figures you can easily work out the amount by which the price is being cut and the price you will actually have to pay.

The first thing to do when you are writing a program is to decide *exactly* what you want it to do. In this case the program will:
(1) take two numbers which you type in, old price and percentage;
(2) determine what sum (in pounds) is equivalent to the percentage of the price;
(3) subtract the discount from the old price to work out the real price, and
(4) tell you what the results are, and ask you if you want to do the same with any other prices.

That's simple enough. The next stage is to decide how the program should do the job. There are always various ways of putting a program together, but there are some useful general principles.

MODULAR PROGRAMS

A tidy, well-written program should be *modular*—with clearly structured and interconnected parts. The reason for this is that you will definitely need to understand the program once you've written it. There's no point in writing line after line of BASIC if you want to change something later and can't understand the program.

The best way to look at it is that your main aims in writing a program are to make it work correctly with as little effort as possible and to be able to understand it quickly once you've finished. The more clearly you write the program, the easier it will be to sort out both the initial problems and the shortcomings of the program which become evident once it has been in use for some time.

The order of a modular program looks essentially like this:
(1) global variable declarations;
(2) the main (controller) program;
(3) an END statement (which marks the point where the computer will stop running the program; it need not be the final line in numerical order);
(4) functions which handle parts of the main task;
(5) DATA statements if used (these can also come towards the beginning).

Following a structure like this will make it much easier for you to write a complex and efficient program. It will make it less time consuming (and at times frustrating) if, for instance, you know exactly where to put, and later look for, a function.

THE PROGRAM ELEMENTS

Global variables (i.e. those used throughout the program) should come first; this is because, if the computer reaches a line containing a variable which you haven't declared, it may stop working through failure to understand the program. So if you declare them all together at once and right at the start, you won't have to worry about them any more. You also have all the important variables together at once, so you know which one is which.

The other type of variable is local, which is one used in a function and only within that function. Those do not need declaring until the function is written—and then they should be at the beginning of it.

The main program should then come second, for two reasons. In the first place, a program is easier to write if the part which controls all the other parts appears towards the beginning. Secondly, you can look near the top of the program, even some time after you've written it, and just read the main program to remind yourself of what it all does. The main program will also use some, if not all, of the global variables, which should be declared before they are used.

The main part of the program calls sub-programs and values from functions. A function can be thought of as a black box. The main program puts numbers or strings into the black box, which then feeds new ones back.

Each function can also be dealt with as a separate program. It can contain its own functions at a lower level. This allows programs to be designed with a clearly understood hierarchy.

DATA statements don't necessarily need to come at the end of a program, but they all ought to be in the same, fairly obvious, place. It can be fiddly to search through a lot of them or count how many there are. Putting them all together saves time spent searching for them if one is wrong.

DATA statements can also be used within a function, as long as they are local to that function.

The END statement, of course, marks the end of the main controller program. As mentioned earlier in this chapter, it is not always the last line of a program. Where functions or subroutines are used, the computer 'jumps out of' the main program to use them, rather than working through in sequential order. The END statement should therefore be in the place where the controller program would actually finish.

Lastly, remember to include plenty of comment lines in all your programs, broken up so they are prominent. Some of the programs in this manual contain far more comments than lines of BASIC. The comments are there to make sure you know exactly what the program is doing. They allow you to explain in English anything which might be obscure when you look back at the program at a later date.

Please don't think that programming is all about obsessive neatness. It isn't. But just as in learning to read and write you had to get used to certain disciplines, so you need to in programming as well. It will help you a great deal in the long run if you remember to apply this chapter whenever you put a program—however small—to paper and then to keyboard.

THE EXAMPLE

Let's return to that price problem now.

First, you need to decide what the main program will do. In this case it simply asks you to type in the price of the goods and the percentage reduction, passes those values to the function part of the program, and tells you the result, finishing off by asking you if you want to do any more.

Next, work out how many functions you will need. In this case you only need one, to handle both the 'how much is the discount' and the 'how much will it cost'—they are both very simple calculations. Let's call the function DISCOUNT.

The next step is to work out how the computer would perform the calculations.

In this program, you will give the computer a number—100 for instance, and a percentage—let's say 20%.

First, the computer has to work out what 20% of 100 is.

It begins by finding out 1%—which is one hundredth—and then it multiplies that by the percentage. So the calculation involved would simply be:
(price/100)* percentage

Then the machine needs to work out what the goods would cost you. It does this simply by subtracting the discount from the original price.

Now you should have some idea of what variables you will need. They are (complete with some names):

- The original price—PRICE
- The discount percentage—DISC
- The discount in pounds—TOTDISC (total discount)
- The discount price—NEWPRICE

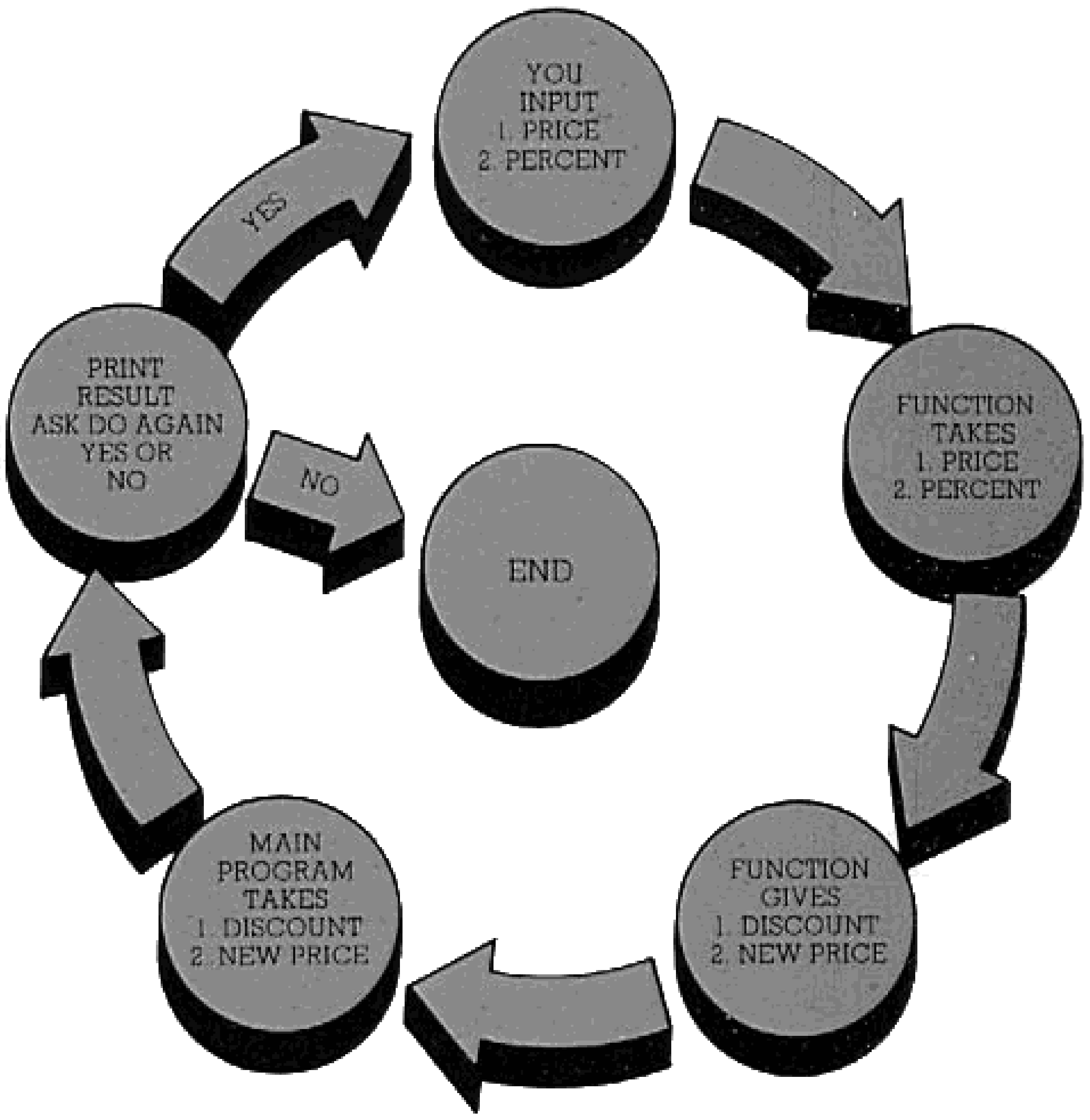
The first two variables will be entered in response to an INPUT statement. So you don't need to declare them. Now you know that you only need to declare two global variables at the beginning.

The function can be written immediately. You don't have to tell the computer you're working with percentages—it wouldn't understand you. A percentage is only a certain number of hundredths.

```
DEF DISCOUNT
  LET TOTDISC=(PRICE/100)*DISC
  LET NEWPRICE=PRICE—TOTDISC
END DEF
```

Now you know roughly which parts come in which order—and exactly what you're trying to achieve, you can begin to put the program into BASIC. With a more complex program you would have far more working out to do, and you would probably write a lot of BASIC on notepaper before you began.

Before you type anything into the computer, take a look at this diagram:


YOU INPUT 1. PRICE 2. PERCENT
FUNCTION TAKES 1. PRICE 2. PERCENT
FUNCTION GIVES 1. DISCOUNT 2. NEW PRICE
MAIN PROGRAM TAKES 1. DISCOUNT 2. NEW PRICE
PRINT RESULT ASK DO AGAIN YES OR NO
YES
NO
END

The diagram shows the order in which the actual BASIC will appear, while the arrows show the order in which the program will operate if you follow them from the YOU INPUT box.

First type in the variable declarations:

```
100     NUMERIC TOTDISC, NEWPRICE
110     !       Then the main program:
120     DO
130             CLEAR SCREEN
140             INPUT PROMPT "Please type in the
                price ":PRICE
150             INPUT PROMPT "Please type in the
                percentage discount offered ":DISC
160             CALL DISCOUNT
170             CLEAR SCREEN
180             PRINT "A discount of ";DISC;" percent
                on goods priced at £";PRICE
190             PRINT "Would be £";TOTDISC;"."
```

```
200          PRINT
210          PRINT "The new price would be £";
             NEWPRICE; "."
215          PRINT
218          PRINT
220          INPUT PROMPT "Would you like to
             know more discounts? ": ANS$
230      LOOP WHILE UCASE$(ANS$(1:1))="Y"
240      END
```

That's the main program. Now all you need to do is add the function:

```
250      DEF DISCOUNT
260          LET TOTDISC=(PRICE/100)*DISC
270          LET NEWPRICE=PRICE-TOTDISC
280      END DEF
```

If you RUN this program now, you'll find it works, but it is really only the bones of a program which need some padding. The program below is essentially the same, but far tidier than what you have just typed in. Anything you don't understand should appear in the ! lines (there are plenty of those!).

```
100      NUMERIC TOTDISC, NEWPRICE
120      CLEAR SCREEN
160      PRINT AT 9,5:"A PROGRAM TO WORK OUT
         DISCOUNTS"
170      PRINT AT 11,9:"ON MARKED RETAIL
         PRICES"
180      !   ---------------------------------------
190      !      160 and 170 print a title in the centre
200      !      of the screen, which is neater
205      !      than the top. 220 and 230 ensure that the
207      !      title remains displayed for about 5
208      !      seconds.
210      !   ---------------------------------------
220      FOR X=1 TO 2500
230      NEXT X
240      CLEAR SCREEN
310      !   ---------------------------------------
320      !      360 to 440 are the main program.
325      !      They control the function called
330      !      DISCOUNT. It is through this part
335      !      that you exit the program when
```

```
340     !       you need to.
350     !       ______________________________
360     DO
365             CLEAR SCREEN
370             INPUT PROMPT "Please type in the
                price of the goods ":PRICE
375             PRINT
380             INPUT PROMPT "Please type in the
                percentage discount offered ":DISC
390             CALL DISCOUNT
400             PRINT "A";DISC;"percent discount on
                goods priced at £";PRICE
405             PRINT
410             PRINT "would be £";TOTDISC;"."
415             PRINT
420             PRINT "The new price would be
                £";NEWPRICE; "."
425             PRINT
430             INPUT PROMPT "Would you like to
                know any other discounts? ":ANS$
440     LOOP WHILE UCASE$(ANS$(1:1))="Y"
445     END
450     DEF DISCOUNT
460             LET TOTDISC=(PRICE/100)*DISC
470             LET NEWPRICE=PRICE-TOTDISC
480     END DEF
```

The program above has one small defect. When it asks you if you want to work out any more discounts, it is unable to cope with a mistaken input. It would be quite easy for you to type 't' instead of 'y', which would have the result of ending the program. A good program should make allowance for mistakes by users.

This may not make a crucial amount of difference to your programs now, but it will when they get longer and more complicated. It's very frustrating to make a typing error which either makes the program finish before you want it to, or just crashes it altogether. *Crash is a very expressive word. Crashing a program means making something go so wrong that the computer stops running it and jumps into immediate mode, so that you have to run the program again. This can be caused by all sorts of things, but normally happens because you write a program which tries to do something that is illegal in BASIC. This is known as a fatal error.*

The program below doesn't provide all the answers, but it does show you how you can make sure a program won't end or fail to work because you type in the wrong input.

```
100    DO
110          !
120          !    The whole program is a large DO/
125          !    LOOP. In 480 to 500 you will
130          !    notice another DO/LOOP inside
140          !    the large one. This is fine—as long
145          !    as the one loop is inside the
150          !    other. A loop inside a loop is
160          !    a nested loop. 175 to 290 are
165          !    also a nested loop.
170          !
175          DO
180               INPUT PROMPT "Type in a whole
                  number from 1 to 5: ": NUM
190               !
200               !    180 asks you to give input as
205               !    shown. 290 makes sure it is an
210               !    acceptable number by check-
220               !    ing first that it is not bigger
225               !    than 5 or less than 1, and
230               !    secondly that it is a whole
235               !    number—this is done by
240               !    making sure the variable
245               !    NUM is not bigger or smaller
250               !    than its integer part. If NUM is
260               !    acceptable, the LOOP is not
270               !    repeated.
280               !
290          LOOP WHILE NUM > 5 OR NUM < 1
             OR NUM < > INT(NUM)
300          SELECT CASE NUM
310          CASE 1
320               PRINT "Number 1"
330          CASE 2
340               PRINT "Number 2"
350          CASE 3
360               PRINT "Number 3"
370          CASE 4
380               PRINT "Number 4"
390          CASE 5
400               PRINT "Number 5"
```

```
410         END SELECT
420         !   ________________________________
430         !   Lines 300 to 410 select the
435         !   right response to your number.
440         !   If the number is not 1 the computer
445         !   drops down to the next line and
450         !   checks that, and so on. SELECT
460         !   CASE is dealt with on page 64.
470         !   ________________________________
480         DO
490             INPUT PROMPT "Do you want to
                try another? ":ANS$
500         LOOP UNTIL UCASE$(ANS$(1:1))="Y"
            OR UCASE$(ANS$(1:1))="N"
510         !   ________________________________
520         !   480 to 500 are the nested loop.
530         !   It will keep looping until the
540         !   first letter of your response,
545         !   converted to a capital letter, is
550         !   "Y" or "N". This is the same in
555         !   principle as the check used on the
560         !   variable NUM above, but the
565         !   method is different. 640 tells the
590         !   computer to go back to the
600         !   beginning as long as the first
610         !   letter of ANS$ is "Y". That means
615         !   that when it is "N" the machine
620         !   leaves the loop.
630         !   ________________________________
640     LOOP WHILE UCASE$(ANS$(1:1))="Y"
650     END
```

Remember, the method used here is just one way of checking input. Many of the functions provided by the computer can be used to do this in one way or another. The secret of writing efficient programs lies partly in inventing little devices like those above—among several other things—and so does much of the fun of it too.

Lastly, your programs will be far more pleasant to use if you make them look tidy to the person using them. Making a program 'look tidy' includes clearing the screen whenever it gets full or whenever the program moves from one stage to another—e.g. from input to the output of results.

You can also print words tidily on the screen by using PRINT AT to give margins, and UCASE$ or LCASE$ to tidy up strings which are typed in by the person using the program. Sound effects and colours can be used as signals.

When you've written some really well structured, useful and attractive programs, you'll not only feel satisfied with what you've done, but also extremely accomplished and knowledgeable as well!

MINIMAL BASIC FEATURES

As you know, BASIC comes in many dialects, just as spoken languages have dialects. In the past, computer companies have invented their own BASICs with their own words and ideas. But the core of the language, and its approach, have remained the same.

The Enterprise follows the dialect of Standard BASIC as proposed by the European Computer Manufacturers' Association and the American National Standards Institute.

Standard BASIC is designed to overcome the problems associated with the use of a non-standard language. It therefore makes it possible to transfer programs between the Enterprise and other machines that use Standard BASIC. This BASIC also provides some very powerful features not available in non-standard BASICs.

You may wish to run programs written in non-standard BASICs without having to go through them and change them in detailed and complicated ways. Fortunately, most BASICs follow something called 'Minimal BASIC', which was devised some years ago and adopted by several manufacturers. You will probably hear about Microsoft BASIC, which is one of those which use the Minimal BASIC features.

To help with compatibility between BASICs, the Enterprise provides the major features of Minimal BASIC as well as those of Standard BASIC. This chapter details these features.

BRANCHES

No, this isn't a lesson on how to prune your prize apple trees. But it is quite helpful if you try for a while to think of a program structure as being in some ways like a tree—with main stems and smaller branches.

Some trees have a lot of branches and others have only a few. Similarly, you can either structure your programs so that they move from one task to another sequentially, or (better) you can organise them so you have a 'trunk', and several 'branches' which spring directly from the trunk. Above all, it's best to avoid leaping from branch to branch. These things have to be treated with care.

Two ways of branching within a program are GOTO and GOSUB. The other main method of branching provided on this computer is that of functions, which are dealt with on page 79.

GOTO and GOSUB both make the computer jump to another part of the program from the part it has reached when it comes to either of these words.

GOTO

GOTO uses a line number to tell the computer where to move to next. Like this:

```
100     CLEAR SCREEN
110     INPUT PROMPT "Do you like computers?
        ":SAY$
120     IF UCASE$(SAY$(1:1))="Y" THEN
130          GOTO 220
135          !              GOTO 220 directs the
136          !              program to line 220.
140     ELSE IF UCASE$(SAY$(1:1))="N" THEN
150          GOTO 240
155          !              GOTO 240 directs the
156          !              program to respond to
157          !              a 'NO' answer.
190     ELSE
200          GOTO 110
205          !              This makes the program
206          !              begin again if you do
207          !              not answer yes or no.
210     END IF
220     PRINT "Oh good I'm okay here!"
230     END
240     PRINT "But I'm different. You'll see!"
250     END
```

So GOTO directs the computer to a line number. You can also use it as a loop. If you changed line 250 to:

```
250     INPUT PROMPT"Would you like to do that
        again? ":A$
```

And added line 260:

```
260     IF UCASE$ (A$(1:1))="Y" THEN GOTO 100
```

(and line 270–END), then the program would loop back to the beginning if you responded with 'yes'. This is the same as making a DO/LOOP out of this program, ending in:

```
260     LOOP WHILE UCASE$(A$(1:1))="Y"
```

Although the computer would not be working out what to do in exactly the same way for both methods, the result would be the same.

GOTO is a far more awkward way to split large programs up into separate components than the use of functions, described on page 79.

The clumsiness of GOTO is mostly because it breaks up the normal order of program flow without providing an alternative structure. This is why programs get lost!

GOSUB

GOSUB is similar to GOTO in that it directs the computer to a line number in a different bit of the program, but with GOSUB you expect to return.

Using GOSUB you need to set part of a program aside as a 'unit'—a bit which is meant to do one thing only. Then, at the end of that bit of program, you must add the word RETURN, to tell the computer to go back to the line immediately following the one that contains the word GOSUB.

The bit of the program between the line indicated by a GOSUB command and the RETURN command is called a subroutine. A function can also be a subroutine, using the word CALL.

HOW TO AVOID MISTAKES

If you don't want to go wrong and become confused, please remember two points whenever you use GOSUB/RETURN:

First, a GOSUB must never be used *without* a RETURN. If this happens, the result is a message from the computer to tell you you've made a mistake.

Secondly, even though you have set the subroutine aside as a separate unit, the computer has not. There is no special command to mark the beginning of a subroutine. Because of this the computer may run into the subroutine by mistake.

If this happens, and the computer reads the word RETURN without having been told to GOSUB first, it will stop the program because it will not understand.

The best way to avoid this problem with GOSUB is to put all your subroutines at the physical end of your programs. Then, immediately before they start, put the END statement or a STOP command. This tells the computer to end the program before it has a chance to read the subroutines again.

The following program shows an example of this technique:

```
 80  CLEAR SCREEN
100  PRINT AT 1,5:"POPULATION TO LIVING
     SPACE"
110  PRINT AT 2,9:"RATIO PROGRAM"
120  PRINT AT 4,5:"This program works out the"
125  PRINT AT 5,5:"average ground space"
130  PRINT AT 6,5:"available to each person in"
140  PRINT AT 7,5:"a city, in square yards."
150  PRINT AT 8,5:"Remember—it's only an"
160  PRINT AT 9,5:"average, but if you compare"
170  PRINT AT 10,5:"lots of cities you can soon"
180  PRINT AT 11,5:"see which are overcrowded!"
183  FOR X=1 TO 5000
184  NEXT X
187  CLEAR SCREEN
190  PRINT AT 14,5:"Please type in the following
     information:"
200  PRINT AT 16,5:"A) Name of city,"
210  PRINT AT 17,5:"B) Population,"
220  PRINT AT 18,5:"C) Size of city in square
     miles."
230  FOR X=1 TO 2500
232  NEXT X
235  CLEAR SCREEN
240  INPUT PROMPT "CITY? ":CITY$
250  INPUT PROMPT "POPULATION? ":POP
260  INPUT PROMPT "SIZE? ":SIZE
270  GOSUB 1000
350  CLEAR SCREEN
360  PRINT AT 10,5:CITY$;", at a size of";
370  PRINT AT 11,5:SIZE;" square miles,"
380  PRINT AT 12,5:"and a population of ";POP
390  PRINT
400  PRINT AT 13,5:"would give everyone living
     there"
410  PRINT AT 15,12: SPA
420  PRINT AT 16,5:"square yards of ground space
     to live in –"
440  PRINT AT 17,12:"on average!"
450  PRINT
455  PRINT
460  INPUT PROMPT "Do you want to try any
     more? ":ANS$
470  IF UCASE$(ANS$(1:1))="Y" THEN GOTO 187
```

```
 480    !    ________________________________
 490    !    470 uses GOTO to send you back to
 500    !    type more info if you answer yes.
 510    !    ________________________________
 520    END
 530    !    ________________________________
 540    !    The END is line 520 so that the
 550    !    subroutine is not run into again.
 560    !    This prevents errors.
 600    !    Line 1000 begins the subroutine.
 610    !    ________________________________
1000    LET SQY=SIZE*1760^2
1010    LET SPA=SQY/POP
1080    RETURN
```

So the program above shows you how to use GOSUB. But as you can see, it is far less tidy than a program using a function with CALL—notice for instance that apart from the GOSUB line there is no indication where the subroutine begins.

Programs which use GOTO or GOSUB are more likely to go wrong than programs which use structured loops and functions.

THE 'DIM' STATEMENT

A look back on the chapter about storing larger amounts of information will remind you how you should set aside areas of memory for use as arrays. There is another, less versatile, way to do this.

DIM A(10) will set aside a one-dimensional array with elements numbered 0–10. DIM A(4,4) will set aside a two-dimensional array using the same element-numbering system. You can use (X TO Y) to specify a range of numbers to be given to your array's elements, as you would using STRING or NUMERIC. What is missing, though, is the ability to decide how long each string element may be. When you write very long programs which use several string arrays, you will perhaps realize how useful this can be, because you can use it to conserve memory space.

DIM is only provided for compatibility with other BASICs. Here is a short program to demonstrate it. Note that DIM ARRAY$(9) is the same as STRING ARRAY$(9)— the bottom element is always 0 unless you specify a different number.

```
100     DIM ARRAY$(9)   !   a 10 element array
110     FOR N=0 TO 9
120       READ ARRAY$(N)
130     NEXT N
140     FOR N=0 TO 9
150       PRINT ARRAY$(N);" ";
160     NEXT N
170     DATA This,is,an,array,declared,
180     DATA using,the,DIM,statement,
190     DATA "—simple, eh?"
```

CHANNELS

In order to give more flexibility in the use of the computer and the 'devices' attached to it, the Enterprise makes use of a concept called channels.

A channel is a special pathway which is opened between two parts of the computer. Once the pathway is open, communication can take place between the two parts of the computer simply by specifying the channel number—preceded in instructions by the symbol '£' (or '#' in some character sets).

The BASIC in the Enterprise tries not to make too many assumptions about how you will use the computer. After all, the power of the computer comes from its flexibility in acting according to your wishes. For this reason, all 'input-output' instructions allow you to specify channels if you wish.

However, because you do not want to be bothered with unnecessary commands, the Enterprise uses 'default' channels whenever it can. Normally when you PRINT, for example, you want the words or numbers to appear on the TV screen. So PRINT "Hello" puts the message on the display.

If, on the other hand, you wished to make the message appear on the printer instead (and you had one attached), you could give the command PRINT £1Ø4: "Hello".

When the computer is turned on, it automatically connects channel 1Ø4 to the printer socket. So any message sent to channel 1Ø4 goes on to the printer.

You can also refer to channel numbers by using variable names, so the program itself can choose where to send a message.

```
12Ø  INPUT PROMPT "Please enter a message: ":A$
13Ø  PRINT
14Ø  PRINT "Where would you like the message
     repeated?"
15Ø  DO
16Ø       INPUT PROMPT "Please enter Ø for screen
          or 1Ø4 for printer: ":CHANNEL
17Ø  LOOP UNTIL CHANNEL=Ø OR CHANNEL=1Ø4
19Ø  PRINT £CHANNEL: ! Blank line to screen or
     printer
2ØØ  PRINT £CHANNEL:A$ ! Message sent to screen
     or printer
22Ø  END
```

This program will, of course, only work if you have a printer connected and turned on. But you can try out the same experiment with channel 1Ø1 instead of channel 1Ø4—this will connect you with the usual graphics screen instead. Include the command GRAPHICS in the program to see what is happening.

A point to remember when using channels is that *all* input-output works this way—including such things as the sound generator and connection with the tape recorders. If you are careless in your use of channel numbers then strange things could happen to your computer! These are not likely to cause any permanent harm, but you might have to start your program again or reset the computer.

For more details on channels, refer to the chapter on commands in the Reference Section, particularly under the heading OPEN (page 168).

To deal with errors, to handle the network or certain other devices, and also to deal with some special conditions which are independent of the normal program flow, Enterprise BASIC provides something called exception handlers.

Exception handling operations are a little like functions. But unlike functions, exception handlers are not normally activated by a specific reference in a program such as a CALL.

First, a little word of explanation. An exception is something which happens independently of a program running at the time, but which may affect the program or be made to affect it in some way. The 'stop' key is an exception, because the program does not need to check whether you have pressed 'stop' but it is affected by it. Program errors are also exceptions.

If you run a program with some sort of mistake in it, the computer will, if the error is of a type which will stop the program, respond to it with a short message and a number. In this case the number is what may be used as an 'exception type' to give the computer a key to what it should do.

BASIC also allows for you to make up your own exceptions. These must be numbered from 1 to 999 and they can be used to deal with wrongly typed input (e.g. a number which is too big or small) or another unusual condition recognized by a program.

Let's begin looking at exception handling by causing one within a program and then dealing with it using exception handler statements.

```
50    WHEN EXCEPTION USE INPUT_ERROR
60    !
65    !     50 tells the computer to use the
70    !     handler (see below) if an exception
75    !     occurs. It is valid until the computer
80    !     reaches END WHEN in 145.
90    !
100   INPUT PROMPT "Please type a word: ":
      STRING$
110   IF VAL (STRING$)< >0 THEN
115         CAUSE EXCEPTION 10
120   ELSE
130         PRINT "Your word has been
            accepted."
140   END IF
```

```
145    END WHEN
150    END
160    !    ____________________________________
170    !    260 to 310 is the part of the
180    !    program which copes with an
190    !    exception; in this case exception
200    !    number 10. This is caused if you
210    !    type in a number, not a word.
220    !    You may notice some parallels
230    !    between the exception handler and
240    !    calling functions.
250    !    ____________________________________
260    HANDLER INPUT_ ERROR
270         IF EXTYPE=10 THEN
280            PRINT "That was not a word."
285         ELSE IF EXTYPE< >10 THEN
290            EXIT HANDLER
300         END IF
310    END HANDLER
```

You can probably see that this could be done using other methods. Line 110 actually makes the computer register a 'mistake' according to your instructions.

CAUSE EXCEPTION is there to send the program to a handler in the event of a mistake which the computer would not normally recognize—for instance, a word beginning with a number, as above. Normally the computer would accept numbers as a string as you know, but the use of VAL makes a string which starts with a number into a mistake in conjunction with the statement CAUSE EXCEPTION.

The VAL of a string will be 0 if there is no valid number in a string (except 0) preceding the first letter or other non-numeric character.

EXIT HANDLER is the same as EXIT DO or EXIT FOR but it only relates to the handler block.

EXTYPE is the type number of the exception—in the program it is number 10. The EXTYPE varies according to what has gone wrong and, in the case of a CAUSE EXCEPTION statement, to what number you give the exception to be caused. See page 180, which deals with error messages. Another word, EXLINE, gives the line number where the exception happened.

The previous program illustrated one type of exception handling. In principle this is rather like CALLing a function (using it as a subroutine or module). Notice that this method used the words WHEN EXCEPTION USE, which you can think of as meaning 'in the event of a mistake anywhere, use these handlers'. This command has to be matched with an END WHEN statement.

The Enterprise is able to communicate with other computers using a simple wire connection. The other computers, of course, have to have the same facility (known as the Intelligent Net) if they are going to manage their end of the conversation, but linking up with other Enterprises is no problem.

The advantage of a net is that many computers can be joined up together, but when one computer wants to talk to another the remaining computers stay out of the conversation—just like a telephone system.

Your computer does not have a pre-set number in the same way as a telephone does; when you connect up to the net you must select a number for it, by typing for example:

```
SET NET NUMBER 5
```

The net number can be anything from 1 to 32. You can use ASK NET NUMBER to remind yourself of the number you have selected.

TRANSFERING PROGRAMS

Once your computer has been given a net 'address' number, it is a simple matter to transfer programs between machines.

On the computer which will be receiving the program, type LOAD "NET-Ø:". This will allow you to accept a program sent from any other computer on the net.

Alternatively, if you only want to receive a program sent from one specific computer, type LOAD "NET-n:", where 'n' is the net number of the other computer. For example:

```
LOAD "NET-17:"
```

will attempt to load a program from the computer with net number 17.

Of course, instructions will have to be given on the other computer to send the program. Computer number 17 would have to be given the instruction

```
SAVE "NET-5:"
```

This would send the current program on computer 17 to computer 5.

BROADCASTING

Net number Ø is a special case. This number cannot be given to any computer, but is used for general net operations.

For transmitting onto the net, number Ø is used to signify 'broadcast' messages. These are messages which are not directed at any particular computer, but which are broadcast (like a radio signal) to any computer which is turned on and listening.

This facility is very useful if you have a short message for all other computers, or if you do not know the numbers of other computers on the net.

One problem with broadcast transmissions is that they are not a very reliable means of communication. With a directed signal, sent to a particular computer, the message is re-transmitted until the receiving computer acknowledges that the message has arrived safely. With broadcast signals, this is not possible. It is also not possible to automatically slow down the speed of transmission to a rate which matches the ability of the receiving computer to deal with the information.

So, the instruction

```
SAVE "NET-Ø:"
```

will send a program for all computers to hear, but there is a strong possibility that the program will not be received correctly. Also the sending computer will not know whether or not the transfer has been successful.

OPEN TO ALL MESSAGES

Net number Ø can also be used for indiscriminate receipt of messages on the net. When a computer channel (see page 132, and OPEN command in Reference section, page 168) is opened to "NET-Ø:", this then becomes a 'general' channel for net operations.

Messages which are broadcast from any other computer on the net will be received if a general channel is open. Also a message which has been directed specifically at your computer will be received by you, even if you had not prepared yourself for this message by opening a special channel for communicating with the other machine.

This does not mean that you will receive messages which are private between two other machines on the net. In a directed message between two computers, no other computer can listen in.

Note that if you have a special channel open to another machine, you will receive messages via this channel (not via the general channel) even if they are broadcast.

COMMUNICATION CHANNELS

For detailed use of the net, a channel is opened specifically for communication with one machine. This is done using the normal conventions for opening channels.

```
OPEN £11Ø:"NET-17:"
```

will open channel number 11Ø for two-way communication with computer number 17.

This channel can thereafter be used with normal input/output instructions, eg

```
PRINT £11Ø: "This is message for computer 17"
LINE INPUT £11Ø:A$ ! A$ will receive line from
computer 17
```

Because the messages sent between computers on the net are buffered (held in memory before transmission, or after receipt) it is often necessary to make use of a couple of special instructions.

FLUSH £chan will force the transmission of any data waiting in a buffer to be sent. Messages will not normally be sent until they are 256 characters long, or the channel is closed, so the FLUSH command should be used whenever a short message is to be transmitted immediately.

CLEAR £chan:NET will clear the input and output buffers. The computer will not accept any message from another computer on one particular channel until its receiving buffer is clear. This is to prevent corruption of data before it has been used by the receiving computer.

If there is data not yet removed from the receiving buffer (by INPUT instruction, for example), and this data can be discarded, then the CLEAR instruction should be used. If there is data not yet sent, use FLUSH before CLEAR.

```
PRINT £11Ø:"Message for computer 17"
FLUSH £11Ø
CLEAR £11Ø:NET
```

```
LINE INPUT £110:A$
```

In selection of channel numbers for the net operations, it is advisable to use a channel number over 100 (but avoiding default channels, see OPEN page 168). This is because BASIC will close all channels 1-99 when it clears variables, which is likely to occur often when in immediate mode.

BACKGROUND NET HANDLING

The most interesting use of the net occurs when a program is written so that net communication can effectively work as a background task during the main operation of a program.

The operating system of the Enterprise uses a method known as 'interrupts' when controlling the net. This means that computers can be talking to each other in their own time, in a way which is invisible to the users of the computers.

BASIC provides a method of dealing with this operation, through exception handlers. Exceptions are errors, or other events, which interrupt the normal course of a program (see Exception Handling chapter, page 134).

When a BASIC program is running, and interrupts from the net have been enabled with the instruction SET INTERRUPT NET ON, then the receipt of a message on a net channel will cause an exception.

An exception handler designed to deal with net operations should always give the instruction SET INTERRUPT NET OFF as the first line in the handler block. This will prevent the computer from becoming confused by receiving a new interrupt while it is within the exception handler.

Once in the exception handler, the net input buffers are 'polled' (checked in turn) by the ASK NET CHANNEL instruction. At any one time, several channels may be holding messages (more messages may come in during the time the handler is in progress). Following the transfer of data from a channel, the ASK NET CHANNEL instruction should be used again until it returns the value 255—signifying that there are no more messages awaiting collection.

SET INTERRUPT NET ON should be the last instruction line before exiting the handler.

Note that exception handlers are triggered by 'software interrupts'. These do not occur in the immediate mode of BASIC, and so a foreground program must be running in BASIC in order to use the exception handler technique for net operations.

The 'brain' of the Enterprise is a Z80 microprocessor. The Z80 can perform around 5ØØ specific operations, each one denoted by a code number—a 'machine code'. If you program a processor in machine code, you are addressing it directly, in its own language, not through the BASIC interpreter.

There are two main reasons why you might want to include machine code routines in your BASIC programs. Either you might need a little extra speed (especially when handling graphics and sound), or you might want to use a feature of the Enterprise's hardware which isn't supported by BASIC.

Machine code programming is a large subject, and cannot be covered in a single chapter of this manual. If you are interested, there are many Z80 programming manuals available.

Enterprise BASIC has several commands which allow you to build up machine code routines (in hex codes) and execute them from within a BASIC program—though these commands are not part of the ANSI standard.

ALLOCATE

First, you need to set aside some memory for storing your code; decide how many bytes long your code will be, then use:

```
ALLOCATE number-of-bytes
```

But note that ALLOCATE destroys all stored variable values, so it's best to use it only at the beginning of a program.

CODE AND HEX$

The machine-code routine is stored—with a given name—by using the CODE command:

```
CODE name = routine in hex codes
```

The name has the same format as a variable name.

CODE can only store a string—your hex codes must be converted into the right format by using HEX$, as follows:

```
HEX$("hex,hex,...")
```

(don't forget the inverted commas, or the commas separating the hex values!); or

```
HEX$ (any string expression)
```

For example, a routine which doubles a specified number—

```
TEST:  29  ADD   HL, HL     ;  add number to itself
       C9  RET
```

—can be inserted into the BASIC program like this:

```
100     ALLOCATE 2
110     CODE TEST=HEX$("29,C9")
```

Once you have run this part of the program, the routine is stored in the memory you set aside.

EXECUTING THE ROUTINE

Broadly speaking, your code will create a new 'function'. In the chapter on defining functions, we divided these into two types. One type is designed to calculate a result—the 'built-in' functions come into this category. A good example is SIN, which calculates the sine of a specific angle when you type something like:

```
PRINT SIN (53)
```

The value to be processed (which you put between brackets) is called the *argument* of the function. Some built-in functions don't take an argument—RND, for example—but all return a value, an 'answer'

The other type of function is more like a 'command'. A command is a set of operations which do something, like clearing the screen or perhaps setting up a graphics mode. A command does not return a value.

The way your routine is executed depends on which of these two types it is. If it returns a value, like our double-the-number example, use

```
USR (name,argument)
```

For example:

```
PRINT USR (TEST, 2)
```

will print 4 on the screen; while

```
LET A=3*USR(TEST, 2)
```

will assign the value of 12 to variable A.

Note that the argument of USR is passed into HL at the beginning of the routine; and the value returned by USR is the contents of the HL register at the end of the routine.

If your routine is a command-type operation, you must use

CALL USR

—which does not return a value. For example:

```
CALL USR(NAME,Ø)
```

You must still give USR an argument, but the value doesn't matter.

'Commands' can be amalgamated as in the following example:

```
CALL USR(CLEAR,Ø)+USR (GRAPH, Ø)
+USR(PICTURE,Ø)
```

WORD$

WORD$ converts its argument into a two-byte string—LSB MSB—so it's useful for forming backward jumps from labels. For example,

```
WORD$(TEST)
```

will return the start address of the TEST routine in the correct format for machine-code jumps and calls.

REFERENCE SECTION

RULES OF BASIC

The Reference Section provides a guide to all the BASIC words available on the Enterprise, along with their purposes and methods of use. Some of them are mentioned only briefly in the Tutorial section, others are not mentioned at all.

It is to be hoped you will experiment with all these words and discover for yourself the full extent of the Enterprise's potential. If you know BASIC already, you will find, in the main, that this section is your best guide to the Intelligent Standard BASIC (copyright Intelligent Software Ltd, 1984) provided on the Enterprise.

GENERAL RULES

Upper and lower case letters are interchangeable in BASIC keywords and identifiers, e.g. FOR, For, for and fOr are all the same word.

A program line may be up to 250 characters long, with a line-number 1 to 9999. It may include several statements, separated by colons; anything permitted after THEN in an IF/THEN statement may be incorporated on a multi-statement line.

An identifier can be up to 31 characters long, and all characters are significant. The identifier can contain letters, numbers, full stops and 'underline' characters; the first character must be a letter.

! is used to mark off the rest of the line as a comment. In immediate mode, a colon at the beginning of a line signifies that the rest of the line is passed through to the operating system without interpretation.

The interpreter deletes spaces before and after the line-number and first keyword, and at the end of the line. It then indents the program for every new block. FOR, DEF, DO, HANDLER, SELECT and WHEN will indent the next line by 2 spaces. ELSE and CASE inside an indented block are placed 2 characters to the right. LOOP, END and NEXT terminate the indentation. Line-numbers are printed with leading spaces to maintain a straight edge.

```
   1 LET A=0
  10 DO WHILE A<10
 100   LET A=A+1
 110   SELECT CASE A
 120   CASE 1
 130     PRINT "first time"
 140   CASE ELSE
 150     PRINT "not first time"
 160       END SELECT
 170       PRINT A
 180      LOOP
 190      GOTO 1
1000      END
```

For further reference on the syntax and conventions of the BASIC, see the Draft Proposal for Standard BASIC from ANSI committee X3J2/82-17.

Keywords are given in **BOLD CAPITALS** in the left-hand margin of the page. The formats of the commands using the keyword are given in normal print, and examples are given in *italics*.

MULTIPLE PROGRAMS

IS-BASIC on the Enterprise gives the facility for several programs to be in the computer at one time. Each program has its own line numbers and its own variables.

A program can be referred to either by number, or by a name given on a PROGRAM line. See in particular the commands **CHAIN, EDIT** and **PROGRAM**.

At any particular time, one of the programs (by default, program Ø) is the 'current' one, on which commands such as LIST and RENUMBER will operate. The number of this program is shown on the 'status line' at the top of the screen.

Program Ø can use approximately 42 k of memory. Other programs are limited to 32 k each.

EXTENSIONS

The facility exists to provide extensions to BASIC, which may be either loaded from cassette or disk or included in an add-on stack unit for the computer. Explanation of the extra commands or functions will be provided in the instructions accompanying such products.

DATA TYPES

Two explicit data types are provided: numeric and string. Numeric variables have names which follow the rules for identifiers (see General Rules). Identifiers for string variables must end in a $ symbol.

Numeric values are calculated in binary-coded decimal arithmetic, and printed to 1Ø digits. Numbers are handled in the range $1e^{-64}$ to $9.999999999e^{62}$

Strings have a maximum length of 254 characters, if declared to this length (see **STRING**). Sub-strings may be referenced in the form string-id (x:y), which specifies a string beginning with character number x, and ending with character number y. If either x or y are omitted, they default to the start or end of the string respectively.

OPERATORS

Arithmetic operators:

*	—	multiply
/	—	divide
^	—	to the power of
+	—	plus
−	—	minus

String operators:

&	—	concatenate

Relational operators:

>	—	greater than
<	—	less than
=	—	equals
> =	—	greater than or equals
< =	—	less than or equals
< >	—	not equals
AND	—	logical AND (true/false)
OR	—	logical OR (true/false)
BAND	—	binary logical AND
BOR	—	binary logical OR

ABBREVIATIONS

The following abbreviations are used in this reference:

chan	—	channel-number
id	—	identifier (e.g. variable name)
str	—	string
var	—	variable
expr	—	expression
relop	—	relational operator (i.e. >, > =, etc.)
para	—	parameter

line-number

line-number text
line-number space
line-number

Adds or replaces a program line. If the line-number is followed by only a space, then a line containing an '!' is inserted. If the line-number is followed by nothing, then the line is deleted. Only executed in immediate mode. Clears variables.

Note that all commands or statements which clear variables also close any open channels in the range 1-99 inclusive (see **OPEN**).

ALLOCATE

ALLOCATE expr

Used in connection with machine code subroutines. Moves up the program source to create a gap of the specified number of bytes, where the user's machine code will go. Sets the location counter to the first free byte in the gap. Note that this destroys all variables, so it should only be used at the start of the program.

ASK

ASK machine-option var

Enquires about some option (e.g. KEY RATE); see 'Machine Options', 'Video Options' and 'Sound Options' sections. Compare also **SET** and **TOGGLE**. The variable will take on the current value of the machine-option.

e.g. *ASK KEY RATE A*

assigns the current keyboard repeat rate to the variable A.

AUTO

Special editing command which prints line-numbers automatically. Only works in immediate mode.

AUTO
AUTO AT 1ØØ STEP 1Ø
AUTO STEP 1ØØ

Default starting line-number is 1ØØ. The default step size is 1Ø. New lines replace old ones with the same line-numbers.

AUTO can be cancelled by pressing 'stop'.

CALL

CALL function
CALL function (para-list)

Used to call a function (either built-in, or defined by DEF), when no result is required from the function.

Any expression following CALL will be evaluated, and the result ignored. *CALL USR (A,B) + USR(C,D)* will therefore call two machine code USR programs.

Can be executed in immediate mode.

CAPTURE

CAPTURE FROM £ chan TO £ chan

Captures input from first channel and substitutes it for input expected from second channel. Input from second channel is locked out until 'stop' is pressed, an end-of-file condition arises on the first channel, or an error occurs. CAPTURE FROM a particular channel can also be terminated by giving £255 (normally invalid) as the TO channel, in a later statement.

CASE

See **SELECT** block.

CAUSE EXCEPTION

CAUSE EXCEPTION expr

Causes an error and assigns it to the category denoted by the expression; user values should be between 1 and 999, since these will never be used by BASIC. The word EXCEPTION is optional.

CHAIN

CHAIN program-number
CHAIN "name" (para-list)

Used for executing BASIC programs from the current program.

Parameters may be passed by value from one program to the other, e.g.:

CHAIN "My_Program" (1, "Fred")

See also **PROGRAM**.

CLEAR

CLEAR £ chan:
CLEAR ENVELOPE
CLEAR FKEYS
CLEAR FONT
CLEAR GRAPHICS

CLEAR £chan:NET
CLEAR QUEUE sound-source-number
CLEAR SCREEN
CLEAR SOUND
CLEAR TEXT

Clears various options. Can be executed in immediate mode.

CLOSE

CLOSE £ chan

Flushes any data in output buffers, closes the channel and frees buffers.

CODE

CODE = string
CODE variable-name = string

Used in connection with machine code subroutines. Copies a string to the position indicated by the current location counter. If a variable is given, this takes the value of the location counter. The location counter is left pointing to the byte following the string, which is assumed to contain the machine code. The variable-name can later be used to call the routine, or to form the destination address of jumps etc.

CONTINUE

As a command in immediate mode, it restarts the program at the next line after a STOP command or press of the 'stop' key.

Used in a program as an exit from an exception handler, it resumes at the statement following the one which caused the exception.

COPY

COPY FROM £ chan TO £ chan

Copies the contents of one channel to a second channel (both channels must be open). The copy terminates on end-of-file, an error, or the 'stop' key. Default input is channel Ø. Default output is channel 1Ø4.

COPY

copies from £Ø to £1Ø4.

COPY FROM £5

copies from £5 to £1Ø4. Requires channel 5 to have been opened.

DATA

DATA data-list

Allows the inclusion of a list of constants, numbers and/or strings, for subsequent READing. See **READ**.

DATE

DATE date-string

Specifies the current date held by the computer. The date is automatically incremented when the current time held by the computer reaches midnight.

The date is specified in the International Standard format YYYYMMDD.

DATE "19850727"

is equivalent to 27th July 1985.

Can be used in immediate mode. See DATE$ function.

DEF

DEF numeric-id = expression
DEF numeric-id(parameter-list) = expression
DEF string-id = string-expression
DEF string-id(parameter-list) = string-expression

One-line function definition:

DEF AVERAGE (X,Y) = (X + Y)/2

DEF block: this is a group of statements that can be called as a function returning a value in the expression, or as a procedure statement. There are several small changes from the ANSI definition. See also **CALL, EXIT DEF, NUMERIC** and **STRING**.

```
def-line
  any number of statements of blocks
end-def-line
```

```
def-line:
DEF numeric-id
DEF numeric-id (parameter-list)
DEF string-id
DEF string-id(parameter-list)
```

end-def-line:
END DEF

If the function is intended to return a value, this value should be assigned to the function name within the DEF block.

```
DEF ANSWER (A$)
  IF UCASE$(A$(1:1)) = "Y" THEN
    ANSWER = 1
  ELSE IF UCASE$(A$(1:1)) = "N" THEN
    ANSWER = Ø
  ELSE
    ANSWER = -1
  END IF
END DEF
```

The scope of variables at any point in a program is dynamic—that is, it depends upon the history of which lines have been executed, and not upon the static layout of the program.

```
100     NUMERIC FRED
110     LET FRED = 1
120     CALL Q
130     PRINT FRED
140     END
200     DEF P
210       LET FRED = 123! This FRED is a global.
220     END DEF
300     DEF Q
310       NUMERIC FRED! A local FRED.
320       LET FRED = Ø
325       CALL P
330       PRINT FRED
350     END DEF
```

In this example, FRED is used both as a global and as a local. When line 21Ø is executed, the FRED at 31Ø gets changed to 123 and not the one at 1ØØ. The program will print 123 and 1. In a static scope language, the program would print Ø and 123; this may happen if the same program is run under a compiler BASIC.

Everything declared within a DEF block is local to that block, and allocated at each first execution of the declaration after the call. Anything not declared may be local or global depending on the history.

It is best to declare all variables at the start of each program or function in order to avoid unexpected results.

```
100     CALL P ! This call of P has I as local to P.
110     LET I=9
120     CALL P ! This call of P changes the global I.
130     END
200     DEF P
210        LET I=6
220     END DEF
```

In order to give consistent results, a line

```
90      NUMERIC I
```

should be added to the program; this will make I global in both calls of P.

The memory used for the storage of local variables is released when a function is exited. This characteristic can be exploited for the efficient use of computer memory—for example, a temporary data array can be within a function.

Almost anything can be passed as a reference parameter. Normally parameters are passed by value, which means that copies are passed to the function and any operation inside the function does not change the external variables. Reference parameters take their type from the actual parameter, and any changes inside the function change the external variables also.

```
100     DEF SWAP (REF A,REF B)
110        NUMERIC T
120        LET T=A
130        LET A=B
140        LET B=T
150     END DEF
200     LET X=99
210     LET Y=23
220     CALL SWAP (X,Y)
230     PRINT X, Y
```

```
prints    23    99
```

Arrays and functions must always be passed by reference.

```
100     NUMERIC A(10)
110     OPTION ANGLE DEGREES
120     DEF P (REF FN, X)
130       PRINT FN(X),
140     END DEF
150     LET A(2)=66
160     CALL P(A,2)
170     CALL P (SIN,30)
```

prints   66   .5

Passing built-in and user functions can be very useful for library software. A graph-drawing function can have the function to be plotted passed as a parameter, a sort function can have the exchange and compare routines passed as functions.

Functions can call themselves recursively.

DELETE

DELETE line-description TO line-description,...
DELETE line-description—line-description,...
DELETE block-name

Deletes lines from the program. Only executed in immediate mode. Clears variables.

DELETE LAST
DELETE FIRST TO 100
DELETE 1 TO 199, 300, 500 TO 9999

Acceptable syntax is to use '-' instead of TO. If the first (or last) number in a range is omitted, it defaults to the first (or last) line of the program.

e.g. *DELETE FIRST-100, 500-LAST*
or *DELETE TO 100, 500-*
for *DELETE FIRST TO 100, 500 TO LAST*

Lines defining a function P can be deleted with *DELETE P*. DELETE on its own will remove all program lines; can be halted with 'stop' key.

DIM

DIM array-list

Declares numeric or string arrays; lower bound defaults to Ø if not specified. One or two dimensions are allowed. Maximum length cannot be specified for a string by using DIM, so the default of 132 characters is used. (Compare **STRING**.)

DIM A(1 TO 1Ø), FRED$(9), B(-7899 TO-789Ø)

Note: all the above have 1Ø elements.

DISPLAY

DISPLAY £ chan: AT a FROM b TO c

Defines a window to display a segment of a text or graphics video page. Screen-row 'a' is the position where the top line of the segment will be placed. Parameters 'b' and 'c' are character-rows on the page which is to be displayed, and define the top and bottom lines of the segment. The numbering of character-rows follows the conventions for text, whether the page displayed is text or graphics. See **PRINT**.

DISPLAY GRAPHICS

Sets up 20 lines as graphics, and displays previous graphics page if one was open (£1Ø1). Does not clear text page.

DISPLAY TEXT

Sets up full screen in text mode and displays full page of text if it was previously open (£1Ø2). Does not clear graphics screen.

If only a small text page was previously open, then this is cleared, and a new full-size text page is opened.

DO

```
do-line
   any number of statements or blocks
loop-line

do-line:
DO
DO WHILE relational-expression
DO UNTIL relational-expression

loop-line:
LOOP
```

LOOP WHILE relational-expression
LOOP UNTIL relational-expression

The structure of a loop is defined as a block, with a DO line, the loop body, and a LOOP line. DO or LOOP cannot be placed on a conditional line.

```
DO WHILE A>3 AND A<1Ø
  LET A=A+1
  PRINT A
LOOP
```

Control cannot be transferred from outside to inside of a loop. See also **EXIT DO**.

EDIT

EDIT program-number
EDIT "name"

Makes the specified program into the current one, so that LIST, RENUMBER, RUN etc. will operate on it. Only works in immediate mode. See **CHAIN, INFO** and **PROGRAM**.

ELSE

See **IF**.

END

Halts execution, marks the end of the program. Also **END DEF, END HANDLER, END IF, END SELECT** and **END WHEN** mark the end of their relevant blocks.

ENVELOPE

ENVELOPE £chan: NUMBER
a;b,c,d,e;f,g,h,i;...;RELEASE;j,k,l,m;...

Defines a sound envelope to be used in conjunction with a controlling SOUND statement. The number 'a' which identifies the envelope must be in the range Ø-254.

Parameters 'b', 'c', 'd' and 'e' define the first phase of the envelope; 'b' gives the change of pitch in semitones (decimal places allowed), 'c' and 'd' specify the change in volume for the left and right speakers respectively, and 'e' gives the duration of the phase, in 'ticks' (one tick is 1/5Ø second).

The values for 'c' and 'd' are in the range Ø-63; they specify the change in volume as a proportion of the overall maximum volume allowed by the SOUND statement. A value of -63 will turn the sound off (any overshoot is ignored); the sound is assumed to be 'off' at the beginning of the envelope. If stereo equipment is not in use, the volume at any moment will be determined by the sum of the values (dependent on SOUND and ENVELOPE statements) for the left and right speakers.

The next phase is defined by 'f', 'g', 'h' and 'i'... For the number of possible phases, see **SOUND BUFFER**, under 'Sound Options'.

RELEASE is optional; it may be followed by any number of phases with their separate parameters. The 'release' phases are performed after the conclusion of the previous phases, or at the expiry of the SOUND duration if there is no following sound on the same channel.

EXIT DO
EXIT FOR
EXIT DEF

Breaks out of FOR, DO or DEF block. Not valid unless inside the right sort of block.

EXIT HANDLER

Breaks out of an exception handler, which propagates the exception to the surrounding environment. This will cause another exception handler to be activated, either a user handler or the default handler.

EXT

EXT parameter-string

Passes a string through to the operating system, which is then passed to valid external programs in memory (either ROM or RAM). The string is then interpreted by these programs as appropriate.

Usually the first word of the parameter string specifies a command to be operated, or a new program to be jumped to.

For example:

EXT "WP"

jumps to the built-in word processor of the computer.

The word "HELP" has a special significance as it requires all external programs to reply with their names. Replies from the external programs are sent to the default system channel. See the **DEFAULT CHANNEL** machine option.

Additional application and service programs will define their own command names and parameter requirements. Often these programs will respond to the instruction

EXT "HELP NAME"

(where "NAME" is the main name of the program) by giving a list of available string commands.

The same effect as EXT can be obtained in immediate mode by starting a line with a colon. In this case, no quotes are required around the parameter string.

:HELP NAME

FLUSH

FLUSH £chan

Forces data remaining in a channel buffer to be sent, without closing the channel or signalling end-of-file. This operation is only appropriate to certain devices (eg NET:). Can be used in immediate mode.

FOR

```
for-line
   any number of statements or blocks
next-line
```

for-line:
FOR simple-variable = expression TO expression STEP expression

STEP can be omitted—the default STEP value is 1.

next-line:
NEXT
NEXT variable

The structure of a FOR loop is defined as a block, with a FOR line, the loop body, and a NEXT line. FOR and NEXT cannot be placed on a conditional line. Allowed in Minimal BASIC.

```
FOR Y=Ø TO 1Ø STEP 2
   PRINT Y
NEXT Y
```

The value of the control variable after the loop has ended is the terminating value plus the STEP expression, i.e. Y will have the value 12 in the example above.

Nested FOR loops cannot use the same control variable. The limit and increment expressions are copied to hidden local memory on execution of the FOR line; these values cannot be changed by the body of the loop. Control cannot be transferred from outside to inside of a loop. See also **EXIT FOR**.

GET

GET £chan: string-id

Gets a single character from a channel, and returns a null string ('' '') if no character is available.

Defaults to channel 1Ø5 (KEYBOARD:), and in simple usage is similar to the function INKEY$.

GOSUB

GOSUB line-number

Calls subroutine beginning at the line-number specified.

GOTO

GOTO line-number

Program execution is continued at the line-number specified. Can be used to exit FOR, DO, HANDLER or DEF blocks, but this is not recommended.

GRAPHICS

GRAPHICS
GRAPHICS HIRES/LORES colour-quantity-number
GRAPHICS ATTRIBUTE

The command GRAPHICS has the effect of closing and re-opening the default graphics and text pages (£1Ø1 and £1Ø2); it displays the default graphics page over most of the screen, but with four lines of text at the bottom.

GRAPHICS also establishes the default channel (1Ø1) for video machine options such as PALETTE.

Valid colour-quantity numbers are 2,4,16 and 256. If nothing is specified for the colour quantity or the HIRES/LORES option, the values that were used for the previous GRAPHICS command will be re-used. For the significance of these values, see 'Video Mode', in the 'Video Options' section. Initially, GRAPHICS selects a high-resolution graphics page with 4 colours.

GRAPHICS ATTRIBUTE selects an 'attribute' mode of graphics in which each colour-cell (8 dots wide by 1 dot deep) can contain one 'ink' colour and one 'paper' colour. This mode combines a 16-colour palette with the same resolution as 4-colour HIRES graphics (resolution and colour-quantity cannot be specified by the user). Both printing and plotting commands may be given, although there can be interactive effects between the colours. For flexible use of this mode see the **ATTRIBUTES** video option. See DISPLAY GRAPHICS.

HANDLER

```
HANDLER handler-name
  exception handler statements
END HANDLER
```

The HANDLER block is used for dealing with program exceptions caused by errors, the CAUSE EXCEPTION command, or machine interruptions.

The handler to be used is specified by the handler-name given in the current WHEN block.

See **CONTINUE, RETRY, EXIT HANDLER,** and the functions **EXLINE** and **EXTYPE.**

Control can be transferred into an exception handler only as the result of an exception (not by a GOTO or GOSUB).

If an exception occurs inside the exception handler the effect is similar to EXIT HANDLER, since control passes to the next outer level of handler (as specified by the next outer level of WHEN block). However, the former values of EXTYPE and EXLINE will have been replaced by new ones.

IF

```
IF relational-expression THEN line-number
IF relational-expression THEN simple-statement
```

Statements not allowed on an IF line are DATA, DEF, END, DIM, NUMERIC, STRING, a further IF, or any statement which introduces a block.

IF A> =3 AND A< =9 THEN 1ØØ
IF A> =3 THEN GOTO 1ØØ

```
if-line
  any number of statements or blocks
else-if-lines option
  any number of statements or blocks
```

else-line option
  any number of statements or blocks
end-if-line

if-line:
IF relational-expression THEN

else-if-line:
ELSE IF relational-expression THEN

There can be any number of ELSE IF lines.

else-line:
ELSE

end-if-line:
END IF

IF blocks can contain any statement which is not restricted to immediate mode.

```
IF A<10 THEN
  PRINT A
ELSE IF A>30 AND A<=40 OR A>50 THEN
  PRINT A+100
ELSE
  PRINT B
END IF
```

The ELSE and ELSE IF lines can be used to break the block into sub-blocks with the usual meanings. ELSE may only be used once, but ELSE IF can be used as often as needed. Control cannot be transferred from outside to inside of an IF block.

IMAGE

IMAGE: format-specification

Used in conjunction with PRINT commands, to control the format of the output. The format-specification is a number of characters which, in this context, have the following meaning.

Numeric format characters:—

	— prints a comma in the number.
$	— prints a floating dollar-sign preceding the sign.

–	—	prints a floating space or '–' sign.
+	—	prints a floating '+' or '–' sign.
%	—	prints a digit, including leading zeros.
£	—	prints a digit or space, trailing zeros after a decimal point.
*	—	prints a digit or leading '*'.
.	—	prints a decimal point.
^	—	prints exponent part; minimum 4 characters.

If the number does not fit in the format space, an error is generated.

String format characters: —

<	—	left-justification of the string, in the field defined by '£' characters.
£	—	prints a character.
>	—	right-justification of the string.

The 'justify' format character must start the field; if no 'justify' character is used, the string is centred.

The format in the IMAGE line starts immediately after the ':' and ends with the last printed character on the line.

INFO

Prints out the amount of memory in the system and the number of unused bytes. A table of information about the programs in memory is also printed, in the following form:

program-number	number of bytes in program	first line of program

INFO clears all variables. Only executed in immediate mode.

INPUT

INPUT £chan, IF MISSING action, AT row-expr, column-expr, PROMPT string: variable-list.

Reads data from channel into a list of variables. Default channel is the editor (channel Ø). Items of data read in to match with variables in the variable-list must be separated by commas.

```
INPUT PROMPT K$&"Enter next number please? ":N
INPUT A(I), B$
```

The IF MISSING and PROMPT parts can be in either order, or absent. The default input prompt is "? " PROMPT replaces the default prompt with the string.

The AT option (with row-expr, column-expr) is independent of the PROMPT option.

IF MISSING is used if an end-of-file condition occurs on the channel, or if there is null input when a numeric input is expected. The action then taken follows the same rules as with READ.

See also **LINE INPUT**.

LET

LET variable-list = expression

Simple assignment; LET is optional unless the variable name is the same as a keyword. Listing or saving the program causes the LET to be inserted so that the program conforms to the standard. Can be executed in immediate mode.

One value can be assigned to several variables:

```
LET A, B(4), C=Ø
A_VAR=A_VAR+1
A$,FRED$="He said"&"Don't"&". "&FRED$(I:J)
LET INPUT=3
```

LINE INPUT

Similar to INPUT, but reads a whole line (including commas, etc.) for each item in the variable-list—which may only contain string variables.

LIST

LIST £chan:line-description TO line-description
LIST £chan:line-description — line-description
LIST block-name

Lists all or part of the program. Can be stopped by 'stop' key or paused by 'hold' key. Only executed in immediate mode. The default channel is £Ø.

```
LIST 3ØØ
LIST 3ØØ TO 4ØØ
LIST FIRST TO 9ØØ, 1ØØØ, 2ØØØ TO LAST
LIST TO 5ØØ, 7ØØ TO
LIST MY_FUNCTION
LIST LAST
```

TO may be replaced with '-'. Compare **DELETE**.

e.g. *LIST FIRST-100,500-LAST*
for *LIST FIRST TO 100, 500 TO LAST*

LLIST

LLIST list-expression

Identical to LIST, but defaults to £104, the printer listing channel.

LOAD

LOAD £chan:filename

LOAD device-name

Loads a file from the given channel, or, if no channel is specified, from channel 106 (cassette, or disks if attached). If LOAD is typed without parameters it defaults to the 'boot' file (on tape, this is the first file found).

LOAD
LOAD "My_Program"
LOAD "NET-0:"

If the file contains a BASIC program then this replaces the current program in memory. If multiple BASIC programs have been saved as one file, these will replace all programs in memory and return to program 0.

The file can contain other data and program types (such as extensions or other applications programs) which will be handled automatically by the operating system.

See **OPEN** for a definition of a device-name and a filename.

Clears variables. Only executed in immediate mode, but see **RUN** which can be used in a program statement.

LOOK

LOOK £chan AT x,y:v

Assigns to variable 'v' the palette colour at point (x,y) on the standard graphics page or other page specified by the channel expression. Both the channel expression and the AT part are optional. If the AT part is omitted, the current beam (cursor) position will be used. Note that the use of AT will turn off the beam and

	move it to (x,y).
LOOP	See **DO**.
LPRINT	LPRINT print-expression Identical to PRINT, but defaults to £1Ø4, the printer listing channel.
MERGE	MERGE £chan:filename Merges the file from disk, tape or other channel with the current file. Lines from the new program will replace lines of the same number in the current file. Only executed in immediate mode. Clears variables.
NEW	Deletes all the current program. Only executed in immediate mode. Clears variables.
NEW ALL	Deletes all programs from computer memory, and returns to program Ø.
NEXT	See **FOR**.
NUMERIC	NUMERIC variable/array-list Declares numeric variables or arrays (which are local if declared within a DEF function). The default lower bound will be Ø. Compare **DIM**. *NUMERIC I,A(1Ø),B(-1Ø TO 2Ø, 2 TO 4)*
ON	ON expr GOTO line-number-list ON expr GOSUB line-number-list Evaluates expression, converts result to an integer, and uses integer result N to choose Nth line-number from the list (the count starts from 1). Program execution then resumes from that line. If there is no Nth line-number, no action is taken. Use SELECT or IF block for a more readable program. *ON A +2 GOTO 1ØØ,2ØØ,3ØØ,4ØØ,99,7ØØ*
OPEN	OPEN £chan:NAME device/filename ACCESS mode OPEN £chan:device/filename

The access mode is either INPUT or OUTPUT. ACCESS OUTPUT attempts to create a new file (if on tape or disk); ACCESS INPUT attempts to use an existing file. For devices such as VIDEO:, either can be used. The default is INPUT.

Connects a device, or a file in the case of tape and disk, to a channel. Commands may then read, write or otherwise manipulate data from and to the device (or file) by referring to the channel number.

OPEN £8:"DISK-1:TEST_PROGRAM" ACCESS OUTPUT

Only one device (or file) may be connected to a given channel at any one time, although a single channel may be used to access several devices (files) one after the other.

To disconnect the channel from a device (or file), use the CLOSE command.

Channel numbers range from Ø to 254. (255 is an invalid channel number which is used for special purposes.)

The BASIC system uses several channels as defaults when channels are not specified in statements. These channels are:—

Ø —used for command input and normal text output (e.g. for LIST and PRINT). This channel is connected at reset (or power on) to the device "EDITOR:".

The device "EDITOR:" itself uses the devices "KEYBOARD:" and "VIDEO:", set up in video-mode Ø with page-size 24, 4Ø.

This channel is the default assumed for COPY FROM and REDIRECT FROM.

Channel Ø is automatically opened at reset, and remains opened until explicitly closed.

Note that channel Ø is specified as the default command channel for ANSI compatibility. Other default channels are numbered over 1ØØ to leave simple channel numbers available for user definition.

1Ø1 —used for graphics input and output statements. This channel is connected at first use of GRAPHICS command to device "VIDEO:", which is set up in video-mode 1, video-colour 1, with page-size 2Ø,4Ø. Channel 1Ø1 remains open until explicitly closed, e.g. by a TEXT command.

1Ø2 —the standard 'text' page. Automatically opened at reset, with page size 24,4Ø.

1Ø3 —used for standard sound output. The channel is connected at reset to device "SOUND:".

Channel 1Ø3 is automatically opened at reset, and remains opened until explicitly closed.

1Ø4 —used for assumed 'hard-copy' operations. This channel is connected at reset to device "PRINTER:". It is the default channel assumed for COPY TO and REDIRECT TO.

Channel 1Ø4 is automatically opened at reset, and remains open until explicitly closed.

1Ø5 —used for keyboard operations (connected at reset to device "KEYBOARD:"). Remains open until explicitly closed.

1Ø6 —used for file-based input and output operations. Whenever required, the channel is connected to "DISK-1:" if attached; If disks are not attached, it is connected to "TAPE:".

Standard file operations include LOAD, MERGE and VERIFY.

Channel 1Ø6 is only opened when necessary, and is closed following the completion of every operation—unless an OPEN command has been explicitly given.

1Ø7— used for network operations.

Channel 1Ø7 is only opened automatically by a command which assumes this channel for the default, and is closed following completion of the operation.

Channels 1ØØ-254 remain open unless specifically closed, but channels 1-99 are always closed when RUN is typed, or if any other operation takes place which clears all variables. If BASIC discovers a default channel closed, then it will close all channels (Ø-254) and attempt to re-open its default channels. If it cannot do this, BASIC assumes that an unrecoverable error has occurred and flashes the screen border until the computer is reset.

Device names passed through to the operating system are terminated by a colon so that they can be recognized. Where more than one device is known by the same name, a number is appended to the name, e.g. "DISK2:" or "DISK-2:".

The valid names are:—

"DISK-n:"	Disk drives.
"EDITOR:"	Screen editor. This in turn uses devices "VIDEO:" and "KEYBOARD:".
"KEYBOARD:"	Transparent keyboard. Includes external joysticks.
"NET-n:"	Built-in local net. The number 'n' is the network address (in the range 1-32) of the machine with which communications are being established. If 'n' is Ø, this defines a 'general' channel—used for broadcasting to all machines, and for receiving data without specifying the source machine.
"PRINTER:"	'Centronics-style' printer port.
"SERIAL:"	Serial RS423 I/0.
"SOUND:"	Sound generator.
"TAPE-n:"	Tape drives.
"VIDEO:"	Video pages.

As other devices are attached to the computer, they are likely to define additional names within the operating system.

In most cases, only a device name is required for an

OPEN operation. When file-based input/output is used, a filename must be given.

The full specification of a filename is:

"device-n:name"

—"device" is optional; if it is omitted, the system mass-storage default device will be used—for an unexpanded system, this is "TAPE:".

"n" is the device number, and defaults to 1 if omitted; e.g. "DISK:" would go to "DISK-1:".

"name" is the description of the file within a device. It follows the same rules of format as a BASIC identifier, except that only the first 28 characters are significant.

If no colon is included in the filename, it is assumed that the device name has been omitted. So, for example, "SOUND" is a file on "TAPE:", but "SOUND:" is the sound generator device.

eg, "DISK1:DATAFILE" "DISK-1:DATAFILE" "1:DATAFILE" "DATAFILE" will all reference the same file (assuming disks are attached).

The "name" part of a filename is ignored by all currently-defined devices except "TAPE:" and "DISK:". So, for instance, "PRINTER:PRETTY-LISTING" is equivalent to "PRINTER:".

There are some commands which allow you to specify both channels and filenames within the one statement; e.g. LOAD and SAVE.

The full specification in these cases takes the form:

£chan:filename

If £chan is missing, then a default channel is used.

OPTION OPTION ANGLE DEGREES/RADIANS

Selects the base unit for subsequent operations using angles. The default is radians.

OUT OUT n,a

Writes byte 'a' to the I/O port 'n'.

PING Produces 'ping' sound.

PLOT

PLOT £chan:point-list
PLOT £chan:ANGLE expr
PLOT £chan:FORWARD/BACK expr
PLOT £chan:LEFT/RIGHT expr
PLOT £chan:ELLIPSE expr, expr
PLOT £chan:PAINT

PLOT followed by a point-list plots points and/or lines. When a PLOT command ends in a semicolon, the beam will be left 'on' after the command has been executed, otherwise it will be turned 'off'.
Thus:—

PLOT x, y

will move the beam—drawing a line, if the beam was 'on'—to position (x,y), and then turn the beam off.

PLOT x,y;

will leave the beam 'on'.

The last two statements both plot a point at (x,y). If the co-ordinate pair is followed by a comma, the beam is moved to the specified position without plotting a point there (and is left 'off').

PLOT x1,y1;x2,y2;...

will draw lines with the beam 'on' between the specified points, and leave it on if the command ends in a semicolon. If the beam was 'on' before the command is executed, a line will also be drawn from the previous beam-position to the point (x1,y1).

Plotting is done in the current ink colour and according to the current line style and line mode (see the Video Options section).

The co-ordinates used in PLOT statements follow the conventions for GRAPHICS plotting. The bottom left-hand corner of the video page is (Ø,Ø). In the co-ordinate specification (x,y), x is the horizontal position counting from the left, and y is the vertical position counting from the bottom.

ELLIPSE plots an ellipse with its centre at the current beam position. The two parameters that follow give the horizontal and vertical distances from centre to circumference, in graphic screen positions. The ellipse must be plotted with the beam 'off' if a dot is not to appear in the centre.

PLOT 300,350, ELLIPSE 200,300,

will avoid plotting the dot.

PAINT fills an enclosed area (that contains the current beam position) with the current ink colour. The area painted is bounded by a continuous line differing in colour from the original colour of the beam position.

If the beam is in a position where a point, of the current ink colour, has been plotted, then PAINT will have no effect, as it will detect a boundary condition immediately. As with ELLIPSE, precautions should be taken to avoid plotting a point.

PLOT 400, 300, PAINT

A PLOT command will by default go to channel 101.

POKE

POKE address, value

Sets the value of the specified Z80 memory location.

PRINT

PRINT £chan, AT row-expr, column-expr:output-list
PRINT £chan, USING line-number:output-list
PRINT £chan, USING string:output-list

An item in the output-list can be either an expression or the word TAB followed by a column-number in brackets. Items may be separated by commas or semicolons. A semicolon generates a null string; a comma inserts spaces up to the start of the next print zone. TAB inserts spaces up to the specified column. An output list ending with a comma or semicolon does not generate an end-of-line sequence. Can be executed in immediate mode.

The AT option positions the cursor at the specified row and columm before printing the list. The optional channel number redirects the output (default channel is the standard text page).

The row and column co-ordinates for the AT specification follow the conventions for text positioning. The top left-hand corner of the video page has text co-ordinates (1,1). The fifteenth column in the second line has text co-ordinates (2,15).

PRINT "VALUE = ";A
PRINT AT x,y:"o";

The USING option controls the format of the output. The line-number must be the number of an IMAGE statement. See IMAGE for the details of the format specification.

PROGRAM

PROGRAM name (variable-list)

Defines the name of the current program, for use in CHAIN statements and as default name for SAVE.

PROGRAM "My_Program" (A,B$)

The variable-list (if included) allows the specified parameters to be passed by value from another program. See **CHAIN** and **EDIT**.

RANDOMIZE

Normally each run of a program starts with the same random number sequence. RANDOMIZE changes the random numbers to a fresh sequence.

READ

READ variable-list
READ IF MISSING line-number:variable-list
READ IF MISSING EXIT DO: variable-list

Reads data from the DATA statements; the IF MISSING action is executed on an attempt to read past the end of the data.

READ A,B$(i)

REDIRECT

REDIRECT FROM £chan TO £chan

Reads input from the first channel and directs it to the second, until the end of a file is reached, the 'stop' key is pressed, or there is an error from one of the channels. The redirection can also be halted by use of the invalid channel number £255 as the TO channel in a later REDIRECT statement.

REM

REM comment-line

Remark line.

REM must be at beginning of line and be followed by at least one space. For greater flexibility, '!' is recommended.

RENUMBER

RENUMBER line-description TO line-description AT expr STEP expr

RENUMBER block-name AT expr STEP expr

Renumbers all or a part of the program. Only executed in immediate mode.

RENUMBER FIRST TO 100
RENUMBER 10 TO 100 AT 300 STEP 10
RENUMBER STEP 100
RENUMBER MY_FUNCTION AT 5000

STEP and AT can be in either order or omitted. If STEP is unspecified, the default is 10. If AT is omitted, then the first line-number in the segment to be renumbered is used. If no line-number range is given, then the whole program is renumbered and the default for AT is 100.

The name of a DEF or HANDLER block can be given instead of a line-number range. For the syntax of the line-descriptions, compare **DELETE**.

All references in the program to renumbered lines are changed.

RENUMBER cannot change the order of lines in a program. So if the renumbered lines would overlay or surround lines not renumbered, or would be put into a new place in the sequence, or would create too high a line-number—then the RENUMBER command is not executed, and the text of the program is left unchanged.

RESTORE

RESTORE
RESTORE line-number

Resets the start of DATA (for READ statements) to the start of the program or the given line-number.

RETRY

Used as an exit from an exception handler, this returns control to the line or statement which caused the exception. Compare **CONTINUE**.

If an exception handler is used to trap the 'stop' key, or any software interrupt, then RETRY should be used to continue the program.

RETURN

Returns from a subroutine called by GOSUB.

RUN

```
RUN (para-list)
RUN line-number
RUN £chan:file-name (para-list)
RUN device-name (para-list)
```

RUN on its own runs the current program from the first line. If a line-number is given, then execution starts from that line-number. If a filename is given (with optional channel), the program is loaded and then run.

Parameters can be passed to programs with RUN, but these must correspond with declared parameters for the program. See **PROGRAM**. Clears variables.

SAVE

```
SAVE £chan:filename
SAVE device-name
SAVE ALL £chan:filename
```

Saves the current program. By default it is saved via channel 1Ø6. If no filename is given, then the **PROGRAM** name will be used, if one exists.

SAVE ALL will save all the programs currently in memory.

Programs are saved in a coded format. To save a program in character (ASCII) format, use LIST £chan:filename—such programs can later be loaded if required. Only works in immediate mode. See **LOAD**.

SELECT

```
select-line
case-line
   any number of statements or blocks
case-line option
   any number of statements or blocks
end-select-line
```

```
select-line:
SELECT CASE expression
```

```
case-line:
CASE expression
CASE expression TO expression
```

CASE IS relop expression
CASE ELSE

end-select-line:
END SELECT

The SELECT block is a group of statements to test the variable or expression against a number of alternative conditions.

The word CASE in the SELECT line is optional unless the expression begins with an identifier CASE.

e.g. *SELECT CASE CASE+23*

There can be any number of CASE lines. The cases are tested in order of line-numbers. There is no point in having additional case-lines after a CASE ELSE, since they cannot normally be reached. Several cases can be combined on one line by separating them with commas.

e.g. *CASE 1,2,3 TO 6, 99*

```
SELECT CASE N
CASE 1
  PRINT "first case"
CASE 2 TO 9,11,21
  PRINT "some more cases"
CASE IS< =A+20
  PRINT "even more cases"
CASE ELSE
  PRINT "rest of cases"
END SELECT
```

The CASE ELSE line can only be used once, and must follow all the other CASE lines. The other CASE lines can be used in any order as necessary, the lines in between two CASE lines forming a block. Control cannot be transferred from outside to inside of a SELECT block.

String SELECTs are also available.

SET

Sets current machine-option values. See 'Machine Options', 'Video Options' and 'Sound Options'. Compare **ASK** and **TOGGLE**.

SOUND

SOUND £chan:PITCH expr, DURATION expr, LEFT expr, RIGHT expr, SOURCE expr, STYLE expr, ENVELOPE expr, SYNC expr, INTERRUPT

Provides overall control of a sound. The parameters may be listed in any order.

The number specified by PITCH may be anything from Ø to 127, although good results are normally obtained only in the range Ø-83. Within that range, an increase of 1 will raise the pitch by one semitone; pitch value 37 (the default) is equivalent to middle C.

DURATION gives the duration of the sound (allocated to the non-release phases of the envelope), in 'ticks' (one tick is 1/5Ø second). The default is 5Ø ticks.

The LEFT and RIGHT parameters specify the overall volume of the sound for the two stereo output channels. The values range from Ø (no sound) to 255 (maximum volume of the machine—the default). If stereo equipment is not being used, the volume will be determined by the sum of the values given for the left and right channels.

SOURCE specifies the tone generator used; the values are Ø-3 (default:Ø). Tone generator 3 is the 'noise generator' (which ignores pitch values).

The STYLE parameter is in the range Ø-255 (default: Ø); for its effects, see the 'Sound Options' section.

ENVELOPE specifies the number of the envelope to be applied to the sound. See the **ENVELOPE** statement. 255 (the default) is a built-in envelope.

SYNC allows the start of the sound to be precisely synchronized with 1, 2 or 3 other sounds from different 'sources'. If, for example, three sounds are to start together, each one can be given the instruction SYNC 2, causing it to be synchronized with the two others. (Default value is Ø).

INTERRUPT, if included, causes the new sound to replace any sound (from the same source) which may currently be going.

SPOKE

SPOKE segment, address, value

As POKE, but writes the value to the system address within the specified segment.

START

If no program is currently loaded, this command loads and runs the first file on channel 1Ø6. If any program is in memory, then START acts as RUN on the current program.

STOP

Halts execution (prints STOP message).
Note that CONTINUE is allowed after a STOP instruction.

STRING

STRING variable/array-list*n

Declares string variables or arrays with maximum length. Default length is 132. Adding *n after the word STRING or the variable declaration sets the length to n. The default lower bound for an array is Ø.

*STRING*8 LAST_NAME$*2Ø,FIRST_NAME$, MIDDLE_NAME$*

In this example, LAST_NAME$ is given a maximum length of 2Ø; FIRST_NAME$ and MIDDLE_NAME$ are up to 8 characters long.

STRING NAME$

Here, NAME$ has a maximum length of 132.

*STRING NAME$ (4 TO 99)*1Ø*

This array has 96 elements, each of 1Ø characters.
Note: a DIM statement cannot be used to define the length of a string variable.

TEXT

TEXT
TEXT 4Ø
TEXT 8Ø

Opens a text page covering the entire display except for the area of the status line. Closes the standard graphics page if it was open.
4Ø or 8Ø specifies the number of columns on the screen. If this is not specified, then the previous value will be used.

See **DISPLAY TEXT**.

THEN

See **IF**.

TIME

TIME time-string

Specifies the current time held by the computer. This is automatically incremented once a second.
The time is specified in the format HH:MM:SS.

TIME "15:35:00"

is equivalent to 3.35 pm.
Can be used in immediate mode. See TIME$ function; see also the command **WAIT DELAY** and the machine option **TIMER**.

TOGGLE

Acts on machine options that have only two possible values (e.g. 'on' and 'off'), by switching from the current value to the alternative. See 'Machine Options', 'Video Options' and 'Sound Options' sections; compare SET and ASK.

TRACE

TRACE ON TO £chan
TRACE OFF

After TRACE ON, the number of the line currently being executed is reported. The output is directed to channel Ø unless directed to a specific channel number.

TYPE

TYPE

Special instruction to exit BASIC, and enter the built-in word processor. All BASIC programs and variables will be destroyed by this action, and so the user is prompted to press 'enter' to confirm the instruction.

VERIFY

VERIFY £chan:filename

Verifies that a program has been saved correctly; compares the current program file with the specified file, and gives an error message if the two files are not identical. Channel 1Ø6 is used by default. Only executed in immediate mode.

WAIT DELAY

WAIT DELAY expr

Causes the program execution to wait for a delay period specified in seconds.

```
WAIT DELAY 60
```

will suspend program execution for one minute.

Maximum delay is 32,767 seconds. See **TIMER** machine option for automatic time-out operation while continuing program execution.

The word DELAY is optional.

WHEN

```
WHEN EXCEPTION USE handler-name
  statements
END WHEN
```

Specifies the exception handler to be used when an exception caused by program execution occurs inside the WHEN block.

The program statements can include additional nested WHEN blocks.

See **HANDLER**.

MACHINE OPTIONS (GENERAL)

Certain system variables and machine functions can be controlled directly from BASIC; these are called machine options. To assign a value to an option, the command SET is used. Where stated, the options listed below may also be handled in conjunction with ASK or TOGGLE.

DEFAULT CHANNEL

SET DEFAULT CHANNEL expr

Specifies the default system channel. This channel value can then be used by service programs which wish to communicate with the user, but which do not know the purpose of currently-used channels.

In particular this channel will be used by programs responding to the HELP command sent via the operating system.

Before having been SET, this channel will be number Ø.

EDITOR BUFFER

SET EDITOR BUFFER expr

Defines the size of the editor's buffer, in 256-byte chunks, for use with editor channels subsequently opened. Can be used with ASK.

EDITOR KEY

SET EDITOR KEY channel-number

Allows the specified channel to be used as the editor's keyboard input, for use with editor channels subsequently opened. Can be used with ASK.

EDITOR VIDEO

SET EDITOR VIDEO channel-number

Allows the specified channel to be used as the text page for the editor, for use with editor channels subsequently opened. Can be used with ASK.

FAST SAVE

SET FAST SAVE ON/OFF

Sets the fast saving speed for tape operations. This speed is approximately 2400 baud, and is the default rate. If fast save is off, the speed is halved.

Loading from tape automatically copes with variations in saving speed.

Can be used with TOGGLE.

FKEY

SET £chan:FKEY key-number string

Sets the function key to produce the specified string each time it is pressed (a null string will cause an exception). The default channel is 1Ø5.

The function keys are numbered 1-16. Numbers 1-8 are the unshifted function keys; numbers 9-16 are the shifted equivalents of keys 1-8.

The function keys are set up with default strings by the system, and re-definition of the keys will remove the default settings. To return all function keys to their default settings, use CLEAR FKEYS.

To create automatic 'enter', use &CHR$(13).

INTERRUPT

ASK INTERRUPT CODE

Asks the software interrupt code for the last interrupt.

SET INTERRUPT KEY ON/OFF

When 'on', causes a software interrupt from any key-press. Can be used with TOGGLE.

SET INTERRUPT NET ON/OFF

Turns on or off the software interrupt caused by receiving data from the network.

SET INTERRUPT STOP ON/OFF

Turns on or off the software interrupt from the 'stop' key. Can be used with TOGGLE.

KEY CLICK

SET KEY CLICK ON/OFF

Determines whether a click is heard with each key-press. Can be used with TOGGLE.

KEY DELAY

SET KEY DELAY expr

Sets the initial keyboard delay before auto-repeat starts, in units of 1/5Ø second. Can be used with ASK.

KEY RATE

SET KEY RATE expr

Specifies the keyboard auto-repeat rate, in units of

1/5Ø second. Can be used with ASK.

NET CHANNEL

ASK NET CHANNEL var

Returns the channel number from which there is data in a network buffer waiting to be read.

When the first byte is read from this channel, NET CHANNEL moves on to pointing at the next channel which needs to be serviced. The value 255 is returned if no more channels have data.

NET MACHINE

ASK NET MACHINE var

This is updated at the same time as NET CHANNEL, and returns the network number of the remote machine. This is particularly important if the data is received on the 'general' channel (NET-Ø:).

NET NUMBER

SET NET NUMBER expr

Sets up the computer's network address number. This must be in the range 1 to 32. This starts up as Ø, which is not valid as a network address.

A net number must be specified before using the network, and should not be the same value as set by any other computer on the network.

REM1

SET REM1 ON/OFF

Controls remote control switch 1. (Also controlled by tape operations.)

REM2

SET REM2 ON/OFF

As above, but for remote control switch 2.

SERIAL BAUD

SET SERIAL BAUD expr

The parameter (in the range Ø-15) determines the baud rate for the RS232 port and the network, according to the code given below. Can be used with ASK.

Ø =>	5Ø	baud	6 =>	3ØØ	baud
1 =>	75	"	7 =>	6ØØ	"
2 =>	11Ø	"	8 =>	12ØØ	"
3 =>	134.5	"	9 =>	18ØØ	"

4 = >	15Ø	"	1Ø = >	24ØØ	"
5 = >	2ØØ	"			

11 = >	36ØØ	baud
12 = >	48ØØ	"
13 = >	72ØØ	"
14 = >	96ØØ	"
15 = >	96ØØ	"

Default baud rate is 96ØØ (value 15).

SERIAL FORMAT

SET SERIAL FORMAT expr

Defines the word format for the serial device driver. The format is controlled by the binary bits in the number, as follows:

BIT	VALUE	EFFECT	
Ø	Ø	8 bits	
	1	7 bits	
1	Ø	no parity	
2	Ø	even parity	ignored if
	1	odd parity	bit 1 is Ø
3	Ø	two stop bits	
	1	one stop bit	

Bits 4 and upwards must be Ø.

Default format is 8 bits, no parity, 2 stop bits. Use of the network will always re-initialise this default.

STATUS

SET STATUS ON/OFF

Turns the 'status line' (at the top of the display) on or off. Can be used with TOGGLE.

TAPE LEVEL

SET TAPE LEVEL expr

Controls the volume level used when saving to tape. Acceptable tape levels are in the range 1-6, with volume doubling for each level. Level 1 is equivalent to about 40 mV peak-to-peak. Default level is 2.

TAPE SOUND

SET TAPE SOUND ON/OFF

Controls transmission of sound from tape input to sound output. Allows direct throughput of music or speech from the tape onto the internal speaker or hi-fi output. Can be used with TOGGLE.

TIMER

SET TIMER expr

Starts a timer which will cause a software interrupt when it counts down to zero. The value is specified in seconds, maximum 255. Setting the value to Ø will stop the timer without causing an interrupt.

The timer always stops when it reaches Ø, and must be explicitly re-started.

The software interrupt exception (EXTYPE 9229) will have to be dealt with by an exception handler, or the program will stop operation. After the exception, ASK INTERRUPT CODE A will assign to A the value 64 if the interrupt came from the timer.

VARIABLE

SET variable-number, expr
ASK variable-number var
TOGGLE variable-number

Sets, asks or toggles the specified operating system variable. For further details see the Enterprise Technical Manual.

These work on the built-in video device, which can contain many video pages each with different parameters. The commands which work on individual pages can be given a channel specification, but if this is left out, some of them default to the standard text page (£1Ø2), others to the standard graphics page (£1Ø1)—as detailed below.

Note that COLOR is always acceptable in place of COLOUR.

ATTRIBUTES

SET ATTRIBUTES expr

Sets a special flag to control operations in attribute video mode (number 15). Values of this flag have the following significance:

1	—plotting in Øs (paper colour)
2	—plotting without affecting bit map (pixel) data
4	—plotting without affecting ink attributes
8	—plotting without affecting paper attributes
16	—printing in Øs
32	—printing without affecting bit map
64	—printing without affecting ink attributes
128	—printing without affecting paper attributes

To achieve combination effects, the numbers should be added together. The default value is Ø.

BEAM

SET £chan:BEAM ON/OFF

The current graphics plotting position is called the 'beam' position. Whenever the beam is moved, it may or may not leave a line behind it, depending on whether it is 'on' or 'off'. Channel number defaults to £1Ø1.

BIAS

SET £chan:BIAS colour-code

Establishes which group of colours will figure as numbers 8-15 within the palette. The number specified in the command is the standard code-number of any colour within the desired group; there are 32 effective values. The bias may also be specified using the RGB function.

SET BIAS RGB (Ø,.6,.4)

The channel number defaults to £1Ø1. The bias is, however, applied to every palette used on the display.

BORDER

SET £chan:BORDER colour-code

Changes the border to the colour corresponding to the specified standard code-number. Channel number defaults to £1Ø1.

CHARACTER

SET £chan:CHARACTER n,a,b,c,d,e,f,g,h,i

Defines the pattern of the character with ASCII code 'n'. Each of the parameters a-i defines one row of the pattern, starting from the top.

To assist in creating characters, the BIN function can be used to specify each pixel in a row as a Ø or 1.

Although a channel number is specified, the command will affect all video pages. The channel number defaults to £1Ø2.

To return all characters to their default settings, use CLEAR FONT.

COLOUR

SET £chan:COLOUR palette-number, colour-code

Sets the value of one colour in the palette (see PALETTE below). Palette numbers are in the range Ø to 7. The colour code is from the standard range Ø to 255 (or specified with the RGB function).

CURSOR

SET £chan:CURSOR CHARACTER code
SET £chan:CURSOR COLOUR palette-number

Specifies the ASCII code of the character, and/or the palette-number of the colour, to be used for the cursor. Channel number defaults to £1Ø2.

INK

SET £chan:INK colour-number

Sets the current plotting colour. The colour number is a palette-number except in colour mode 3 (256 colours), when it is a standard colour-code number. Channel number defaults to £1Ø1.

LINE MODE

SET £chan:LINE MODE parameter

Determines the interaction between the colours on the existing display and the new lines which are plotted. In mode Ø (the default mode), a new line overwrites anything plotted before. In modes 1-3, the colour used for any part of the new line will be determined by combining the palette numbers of the old and new ink-colours, in the following ways:

mode 1—'or'
mode 2—'and'
mode 3—'exclusive or'

Channel number defaults to £1Ø1.

LINE STYLE

SET £chan:LINE STYLE parameter

The current line-style may be set to any value in the range 1-14, enabling various types of broken line to be plotted. Channel number defaults to £1Ø1.

PALETTE

SET £chan:PALETTE a,b,c,d,e,f,g,h

Sets the values of the first 8 colours in the palette, which are then used by video options such as SET PAPER and SET INK. Channel number defaults to £1Ø1.

Only the first four colours can be used in colour-mode 1, and a graphics page in colour-mode Ø can only use the first two. If only the first 2 or 4 colours are specified, the remainder default to colour Ø.

The colours to be placed in the palette are specified by standard colour-code in the range Ø-255, or by the RGB function (see 'Built-in Functions and Variables'). The 'Teletext primary' colours can be specified by name (e.g. MAGENTA).

The palette contains 16 colours in all, although only the first 8 can be chosen entirely freely. See the BIAS option for details on the remaining 8 colours.

PAPER

SET £chan:PAPER:colour-number

Selects the colour which will be used as a background for printing or plotting. In colour-mode 3, the paper colour is defined by a standard code-number; in other modes, by a palette-number. The channel number defaults to £1Ø1.

For a graphics video page (modes 1 and 5—see **VIDEO MODE** option), the PAPER command will only take effect when the page is cleared—when a new background is selected for the graphics display.

For an 8Ø-column text page (video mode 2), the valid paper colours are palette numbers Ø, 2, 4 and 6. These are paired with ink colours 1, 3, 5 and 7 respectively; a character printed in a specific ink colour will automatically be given the associated paper colour for its own individual background. A colour-pair for ink and paper is selected by typing SET PAPER or SET INK, followed by either of the two relevant palette-numbers.

A 4Ø-column text page (video mode Ø) is similar except that there are only 2 available colour-pairs.

SCROLL

SET £chan:SCROLL ON/OFF

Turns automatic scroll on or off. Channel number defaults to £1Ø2.

SET £chan:SCROLL UP/DOWN n,m

Scrolls the screen up or down from line (n-32) to (m-32). Channel number defaults to £1Ø2.

VIDEO COLOUR

SET VIDEO COLOUR expr

Sets the colour-mode for video pages that are subsequently to be opened. (Channel number is ignored.)

When defining a text video page, colour mode Ø must always be selected. For high-resolution graphics pages, the colour modes have the following significance:—

mode Ø — 2 colours; horizontal resolution 64Ø
mode 1 — 4 colours; horizontal resolution 32Ø
mode 2 — 16 colours; horizontal resolution 16Ø
mode 3 — 256 colours; horizontal resolution 8Ø

On a LORES graphics page (using half as much memory as HIRES), the colour quantity for each mode is as above, but the horizontal resolution is halved.

VIDEO MODE

SET VIDEO MODE expr

Sets the video mode for pages that are subsequently to be opened. (Channel number is ignored.)
Parameter values are as follows: —

mode Ø	— 4Ø-column text page (2 colour-pairs)
mode 1	— high resolution graphics page
mode 2	— 8Ø-column text page (4 colour-pairs)
mode 5	— low resolution graphics page
mode 15	— 'attribute' graphics screen

VIDEO X

SET VIDEO X expr

Defines the horizontal size of video pages subsequently to be opened. (Channel number ignored.) The size is specified as a number of character positions in the range 2-42, using the co-ordinate conventions for text pages.

VIDEO Y

SET VIDEO Y expr

As above, only defines the vertical size of the page as a number of character-rows in the range 1-255.

SOUND OPTIONS

These work on the built-in sound generator.

SOUND BUFFER

SET SOUND BUFFER expr

Sets the size of the sound envelope storage area, for a subsequent open to the "SOUND:" device. The expression is the number of phases. Possible values are 1-255; the default is 2Ø. Can be used with ASK.

SOUND STYLE

The values for the STYLE parameter in a SOUND statement (see 'Commands and Statements' section) have the following effects.

On tone channel Ø: —

16 —	Low distortion.
32 —	Medium distortion.
48 —	High distortion.
64 —	Use high pass filter. Tone channel 1 is clock.
128 —	Ring modulation with channel 2.

On tone channel 1: —

As channel Ø, but high pass filter uses tone channel 2; ring modulator uses noise channel (channel 3).

On tone channel 2: —

As channel Ø, but high pass filter uses noise channel (channel 3); ring modulator uses tone channel Ø.

On channel 3 (noise channel): —

1,2,3 —	Use tone channel Ø, 1 or 2 as clock frequency, instead of the standard 31.25 KHz frequency.
4,8,12 —	Select noise frequency from 15, 11 or 9-bit polynomial counters, instead of standard 17-bit counter.
16 —	Substitute a 7-bit polynomial counter for the 17-bit counter.
32 —	Use low pass filter on noise channel, using tone channel 2 as the clock.

64	—	Use high pass filter on noise channel, using tone channel Ø as the clock.
128	—	Use ring modulator with tone channel 1.

To select a combination of sound style options, add together the values for the individual options and specify the resulting number as the STYLE parameter.

SPEAKER

SET SPEAKER ON/OFF

Controls sound output from the internal speaker; SET SPEAKER OFF is used for silencing the machine quickly.

	Trigonometric functions work in degrees or radians (see OPTION statement). Minimal BASIC functions are ABS, ATN, COS, EXP, INT, LOG, RND, SGN, SIN, SQR, TAB and TAN
ABS(X)	The absolute value of a number. This just means removing the sign from it. So *ABS(−9)* would be 9.
ACOS(X)	The angle associated with cosine X, i.e. the opposite of COS. Thus, *ACOS(COS(X))* is X.
ANGLE(X,Y)	The angle between the positive x-axis and the line joining point (Ø,Ø) to point (X,Y).
ASIN(X)	The angle of which X is the sine.
ATN(X)	The angle of which X is the tangent.
BIN(X)	Returns the number corresponding to the given binary representation, e.g. *BIN(11ØØ1)* is 25.
BLACK	The colour black, equivalent to RGB (Ø,Ø,Ø).
BLUE	The colour blue, equivalent to RGB (Ø,Ø,1).
CEIL(X)	Gives the smallest whole number not less than X. In other words, X is 'rounded up' to the nearest whole number. *CEIL(3.45)* would be 4, and *CEIL(−3.45)* would be −3.
CHR$(X)	Returns the character of which X is the ASCII code-number.
COS(X)	The cosine of X.
COSH(X)	The hyperbolic cosine of X.
COT(X)	The cotangent of X.
CSC(X)	The cosecant of X.
CYAN	The colour cyan, equivalent to RGB (Ø,1,1).
DATE$	Returns the current date in the standard format (YYYYMMDD). See **DATE** command.

DEG(X)	Converts X from radians to degrees. DEG(X)=X*18Ø/PI.
EPS(X)	The smallest quantity that can be added to or subtracted from X to make the computer register a change in the value of X.
EXLINE	Returns the number of the last statement that caused an exception.
EXP(X)	Returns the value of e raised to the power of X. The number known as 'e' (2.71828...) is the base for natural logarithms.
EXSTRING$(N)	Returns the message string associated with exception number N. Note that the string starts with a space.
EXTYPE	Returns the category-number of the last exception.
FP(X)	FP stands for fractional part. *FP(1.23)* would be Ø.23, and *FP(−1.23)* would be −Ø.23. FP is the opposite of IP.
FREE	The amount of memory free and available to the current program. This is not the same as the amount of memory free and usable by BASIC (see INFO command).
GREEN	The colour green, equivalent to RGB (Ø,1,Ø).
HEX$(X$)	Returns a string of bytes given the hex values of the bytes in X$. The hex bytes are in upper or lower case and separated by commas, e.g. *HEX$("21,E3,ff")*
IN(N)	Reads a byte from 1/0 port N.
INF	The largest positive number the Enterprise can handle—its idea of infinity. This number is 9.999999999*1Ø^62.
INKEY$	Returns the character from the keyboard if a key is pressed; otherwise returns a null string ("").
INT(X)	The largest whole number not bigger than X. So *INT(3.4)* would be 3, and *INT(−3.4)* would be −4.
IP(X)	The integer part of X. This means that all figures

	following the decimal point are chopped off. *IP(9.9)* would be 9, and *IP(−9.9)* would be −9.
JOY(N)	Gives a value depending on the state of the switches of the specified joystick: 1 — right 2 — left 4 — down 8 — up 16 — fire button Note that the values might be added together if the joystick is pointed diagonally, or if the fire button is also pressed. The joysticks (value N) are numbered Ø to 2, with Ø being the built-in joystick. For the built-in joystick, the space bar is the fire button.
LBOUND(A)	Lower bound of the dimension of a one-dimensional array A.
LBOUND(A,N)	Lower bound of dimension N of an array A.
LCASE$(A$)	Converts all upper case alphabetic characters (capitals) to lower case (small letters).
LEN(A$)	The number of characters (length) of A$.
LOG(X)	The natural logarithm (logarithm to base e) of number X.
LOG1Ø(X)	The logarithm of X to base 1Ø.
LOG2(X)	Logarithm of X to base 2.
LTRIM$(A$)	Removes all spaces which are at the beginning of the string A$. So *LTRIM$(" Hello")* would be "Hello".
MAGENTA	The colour magenta, equivalent to RGB (1,Ø,1).
MAX(X,Y)	Returns the bigger number of X and Y. So *MAX(6,99)* is 99.
MAXLEN(A$)	Gives the maximum length that was specified for a string variable or array.

MIN(X,Y)	As MAX(X,Y), but returns the smaller number.
MOD(X,Y)	X modulo Y. Or, in simpler terms, the integer remainder of X divided by Y. Note MOD(– 1,3) = 2. See REM(X,Y).
ORD(A$)	Gives the ASCII code for the character in quotes, or the ASCII code of the first character of a string variable. ORD stands for ordinal, and means the number associated with the character, in the character-set used by the computer. Since the Enterprise uses ASCII, the ASCII value is returned.
PEEK(N)	Returns the byte at Z80 address N.
PI	The number known as pi. On the Enterprise this is rounded to 3.141592654. Returns the value.
POS(X$,Y$)	Gives the position in X$ (counting the characters from left to right) where Y$ first occurs. If Y$ cannot be found in X$, the result is Ø. By adding a number after the second string (i.e. POS(X$,Y$,X)), you can tell the machine to begin looking for Y$ from a specific place in X$. If X$ is "LONDON" and Y$ is "ON", then POS(X$,Y$) is 2. But POS(X$,Y$,4) would tell the computer to start from the "D" in "LONDON" when looking for "ON", and would give the result 5.
POS(A$,B$,M)	Alternative version of POS. See above.
RAD(X)	Converts X from degrees to radians. RAD(X) = X*PI/18Ø.
RED	The colour red, equivalent to RGB (1,Ø,Ø).
REM(X,Y)	The remainder of X divided by Y. Note REM (– 1,3) = – 1. See MOD(X,Y).
RGB	Returns the machine-dependent colour number equivalent to the specified mixture of red, green and blue colours. R specifies the proportion of red (Ø to 1), G specifies green (Ø to 1), and B specifies blue (Ø to 1). *e.g. SET INK RGB (1/2,1/3,1/4)*
RND	Generates a random number between Ø and 1. For

	practical use, random numbers are multiplied and made into bigger numbers. *INT(RND*1ØØ)* would give a whole (integer) random number between Ø and 99 inclusive. (RND is never 1.)
RND(X)	Generates an integer random number less than X. The largest allowable value for X is 32767.
ROUND(X,N)	Rounds X to N decimal places. *ROUND(1.7668,2)* would be 1.77. *ROUND(−1.7668,2)* would be −1.76.
RTRIM$(A$)	Cuts off spaces from the end of the string. As LTRIM$, but removes spaces from the right.
SEC(X)	The secant of X.
SGN(X)	Returns the sign of X. Returns −1 if X is negative, Ø if X is Ø, and 1 if X is bigger than Ø.
SIN(X)	The sine of X.
SINH(X)	The hyperbolic sine of X.
SIZE(A)	The number of elements in the array A.
SIZE(A,N)	The number of elements allowed in dimension N of the array.
SPEEK(S,N)	As PEEK, but returns the byte at system address N within the segment S.
SQR(X)	The square root of X. X must be positive.
STR$(X)	Converts value X into a string of digits without leading or trailing spaces, but with a '−' sign if X is negative.
TAB(X)	Only allowed in PRINT statements. Moves the cursor position to column X of the current row.
TAN(X)	Tangent of X.
TANH(X)	Hyperbolic tangent of X.
TIME$	Returns the current time in the standard format (HH:MM:SS). See **TIME** command.

TRUNCATE (X,N)	Cuts N decimal places from X.
UBOUND(A)	Upper bound of the dimension of a one-dimensional array A.
UBOUND(A,N)	Upper bound of dimension N of an array A.
UCASE$(A$)	Converts all letters in string A$ to upper case (capitals).
USR(N,X)	Calls an address N (which will probably have been defined using CODE), and passes the integer X in HL to the machine code routine. The value left in HL will be the value returned by USR.
VAL(A$)	Converts a string to a number (i.e. the opposite of STR$). VAL starts converting at the first digit in the string, and stops when it gets to the first non-digit character.
WHITE	The colour white, equivalent to RGB (1,1,1).
WORD$(N)	Returns a two-byte string containing the upper and lower bytes of N, which is assumed to be an address. N will usually be an address defined by a CODE statement, and allows backward jumps etc. to be formed using labels. The first byte of the string will be the LSB.
YELLOW	The colour yellow, equivalent to RGB (1,1,Ø).

EXOS is short for Enterprise eXpandable Operating System. An operating system is a program that attempts to enable the best and easiest possible use to be made of a computer and its facilities. It forms an interface between high-level programs (such as the BASIC language) and the computer.

The main facilities of a computer are its devices and peripherals. These are such things as the screen, the tape interface, a printer and so on. Thus the main part of an operating system handles the devices and peripherals: the input/output system. Other facilities handled by the operating system include the sharing of memory.

INPUT/OUTPUT SYSTEM

The Enterprise microcomputer is extremely complex; to perform even simple functions like printing a string on the screen requires thousands of machine-level instructions, and to print the same string on a printer requires hundreds more instructions. The Enterprise operating system rationalizes the interface between a program and the microcomputer, making it as easy to print a string on a printer as it is to print a string on the screen. This is achieved by allowing programs to treat all input and output devices in an identical fashion. All input and output is performed through 'channels' (a channel simply connects the program to a device). The channels are numbered from Ø to 254. The operating system provides the following functions on channels:

Code number	Function
Ø	System reset
1	Open a channel (connect a device)
2	Create and open a channel
3	Close a channel (disconnect)
4	Close and delete a channel
5	Read a character from a channel
6	Read a block
7	Write a character to a channel
8	Write a block
9	Return the status of the channel
1Ø	Set and read the channel status
11	Perform a special function
16	Read, write or toggle a system variable
17	Capture input from channel to channel

18	Re-direct channel
19	Set default device name
2Ø	Return system status
21	Link device
22	Read system boundary
23	Set user boundary
24	Allocate a segment
25	Free a segment
26	Scan extensions
27	Allocate channel buffer
28	Return error message
29	Load module
3Ø	Load relocatable module
31	Set time
32	Read time
33	Set date
34	Read date

These functions are used by BASIC to provide input/output facilities. They are available for all languages to use, and thus provide a uniform method of communicating with devices. They are also available to the machine code programmer, making it very simple to write programs in machine code.

To call the operating system from a machine code program, a single instruction is required, followed by the code for the function. For example, to open a channel, the following code is needed:

Machine code	Assembler code
F7	RST 3ØH
Ø1	DB 1

The Enterprise operating system provides many more functions than those listed above. A full list of functions and the calling conventions can be found in the Enterprise Technical Manual.

MEMORY USAGE

The operating system is based in Read Only Memory. This means that the program is stored in ROM but still requires RAM space to store its data. The Enterprise is capable of managing a vast amount of RAM and ROM storage.

This storage is manipulated by dividing it up into 'pages' (not to be confused with video pages); each page is 16K bytes long, and there are 256 pages altogether, giving a maximum store capacity of 4M bytes. The Z80 in the Enterprise can only use four of these pages at any one time (rather like reading a book, where you can only see and use two pages at once).

ERROR MESSAGES

Every so often, you are bound to make the odd mistake in a program. It may be difficult to find where the mistake is—or even what it is.

The computer helps you here by providing messages to tell you as much as possible about what's wrong. If you run a program which contains a BASIC error, the computer will stop when it reaches the point at which it can no longer understand the program, and will display a short statement indicating the cause of the problem.

Remember—the computer can't tell you about other kinds of mistake in the same way. If, for instance, you forget about 'operator priority', or think the result of a calculation would be different from what it really is, this may not stop the program from running all the way through—the program will then simply be doing something other than what you thought it should. The computer can only detect errors in the syntax or organization of your BASIC, or problems caused because an action requested by the program is impossible.

```
*** Not understood.
```

If the program is already running when a problem arises which makes it impossible to continue, the error message will contain the relevant line-number; for example:

```
*** Invalid argument to SQR
300 PRINT SQR(Y)
```

You can now move the cursor up and edit line 300. The functions EXLINE, EXTYPE, and EXSTRING$ are supplied to help in the handling of errors and other exceptions. Each error has its own number, which can be referenced with EXTYPE, and for most of these errors a special message will be printed if it is not suppressed.

If an exception occurs that is not covered by one of the built-in messages, then the general number type is printed, together with the exception number e.g.

```
*Overflow error type 1234
```

The general error types are:

0— 999	User
1000— 1999	Overflow
2000— 2999	Subscript
3000— 3999	Mathematical
4000— 4999	Parameter
5000— 5999	Storage exhausted
6000— 6999	Matrix
7000— 7999	File use
8000— 8999	Input-output
9000— 9999	Exos
10000—10999	Control
11000—11999	Graphical
12000—12999	Real-time
20000—20999	Syntax
30000—	System

The specific messages are:

1000	—Unexpected value given
1001	—Overflow in numeric constant
1002	—Overflow in numeric expression
1051	—Overflow in string expression
1106	—Overflow in string assignment (ie. string too long)
2001	—Array subscript out of bounds
3001	—Division by zero
3004	—Invalid argument to LOG
3005	—Invalid argument to SQR
3007	—Invalid argument to ASIN or ACOS
4000	—Error in DEF parameters
4002	—Argument to CHR$ out of range
4003	—Invalid argument to ORD
4004	—Index to SIZE out of range
4005	—Argument to TAB out of range
4008	—Index to LBOUND out of range
4009	—Index to UBOUND out of range
4301	—Error in CHAIN parameters
5000	—Insufficient memory
5100	—Insufficient stack space
5110	—Insufficient extension space
5120	—Insufficient ALLOCATE space

7001 —Invalid channel no.
7003 —Channel already open
7004 —Channel not open
7401 —TRACE channel not open

8001 —Out of data in READ/INPUT
8101 —Numeric data expected
8201 —Invalid USING string
8202 —No format item in USING string
8203 —USING format item too short

9208 —Cassette CRC error
9209 —Editor—load file too big
9210 —Editor—keyboard channel error
9211 —Editor—keyboard channel error
9212 —Editor—video channel error
9213 —Network link already exists
9214 —Network address not set
9215 —Cannot use both serial and network
9216 —Invalid beam position
9217 —Invalid cursor coordinates
9218 —Invalid row number to scroll
9219 —Invalid video page file
9220 —Invalid display parameters
9221 —Invalid video mode
9222 —Invalid video page size
9223 —Sound queue full
9224 —Envelope storage full
9225 —Envelope too big
9226 —Function key string too long
9227 —Protection violation
9228 —Unexpected end of file
9229 —STOP key pressed
9230 —Invalid escape sequence
9231 —Call not supported by this device
9232 —Invalid unit number
9233 —Device already in use
9234 —Invalid special function call
9235 —Invalid date or time value
9236 —End of file module
9237 —Invalid relocatable module
9238 —Unknown module type
9239 —Invalid Enterprise file header
9240 —Unrecognised command string
9241 —Invalid device descriptor
9242 —Unknown EXOS variable number

9243 —Invalid user boundary
9244 —Cannot free segment
9245 —No free segment
9246 —Insufficient video memory
9247 —Insufficient memory
9248 —Channel open error
9249 —Channel already exists
9250 —Device does not exist
9251 —Channel does not exist
9252 —EXOS stack overflow
9253 —Invalid EXOS string
9254 —EXOS function call not allowed
9255 —Invalid EXOS function code

10002 —Return without GOSUB
10004 —No CASE selected
10005 —Program does not exist

20000 —Not understood
20001 —Invalid line number
20002 —Invalid line number range
20004 —Line number does not exist
20010 —Cannot do specified RENUMBER
20020 —Continue not possible
20030 —Identifier expected
20031 —String identifier expected
20032 —Array identifier expected
20034 —Type mismatch
20040 —Variable not initialised
20041 —Identifier declared twice
20042 —Identifier too long
20043 —Missing closing quotes
20050 —Missing end of block
20051 —Invalid end of block
20052 —Too many nested blocks
20060 —Invalid machine option use
20071 —Statement in immediate mode
20072 —Command in program
20073 —Statement not allowed after THEN
20074 —Invalid multi-statement line
20075 —Line too long
20080 —Invalid file format
20081 —Programs do not VERIFY

30000 —BASIC data has been corrupted

Error messages can be trapped, if desired, by using WHEN EXCEPTION and a handler block (page 134). An exception handler can be used to trap any error, even those such as a memory overflow or a syntax error (a keyword mis-spelled, for example). This must be handled with care, as a RETRY to a permanent error will cause the program to loop indefinitely.

Errors like a division by zero, or a negative SQR argument, can be caught without crashing the program.

GLOSSARY

This Glossary is here to help you become familiar with all the computer jargon you will meet as your interest grows. Most of the words within it appear in the manual somewhere, but others do not—the manual has tried wherever possible to avoid jargon and give explanations instead. You will find this Glossary useful if you read computer books or magazines which do not contain a glossary but are full of words you do not understand.

ACCESS	As a verb: retrieving information from an outside storage device—printer or cassette, for instance. Also retrieving information from a program such as database.
ADDRESS	A 'place' inside the computer's memory. Specified using a number, either in *decimal* (the normal counting system), *hexadecimal* (counting to a base of 16) or *binary* (counting to a base of 2). An address inside a computer can contain one of several numbers, depending on what the computer, or the program it is using, is doing. For instance, addresses which control the screen display contain different numbers depending on what is to appear there. 'Address' is also used as a verb when examining the contents of a memory location.
ALGORITHM	The series of ideas and tasks behind a program. First work out your algorithm, then write your program. An algorithm is the system by which a problem is solved.
ALPHANUMERIC	Letters or numbers. The name given to the character set excluding special or graphics characters.
ANSI	American National Standards Institute. The American counterpart of the British Standards Institute. A joint committee of ANSI and the European Computer Manufacturers' Association created the specification for Standard BASIC.
ARGUMENT	See OPERAND and PARAMETER.
ARRAY	A variable which itself contains several more variables. Can be thought of as a list (one-dimensional) or as a grid (two-dimensional).

ASCII	American Standard Code for Information Interchange. A system of coding characters for use within the computer and for transmission from computer to other machines and back. Each character is given a number.
ASSEMBLER LANGUAGE	A programming language which talks to the computer on its own terms. Assembler has simpler instructions than BASIC and, although tedious rather than difficult to use, offers you the capability to program your computer in far more detail than you would using BASIC. Programming in Assembler language is called programming on a low-level. Using it, you would be communicating with the computer in small, very simple keywords, and using codes like ASCII more than you would in BASIC.
BASIC	The programming language supplied with the Enterprise (and most other small computers). There are several different types or dialects of BASIC. The name is an acronym which stands for Beginners' All-purpose Symbolic Instruction Code. Originally designed on huge computers as an aid to learning programming.
BAUD	The rate at which data can be sent out of or into the computer. It roughly equals 'bits per second'. The Enterprise normally loads in a program from cassette at an equivalent of 24ØØ baud.
BINARY	Counting with 2 as a base instead of 1Ø. Binary numbers, therefore, are composed entirely of ones and noughts. Thus 9 would be 1ØØ1. Binary numbers can be viewed as columns, just as ordinary numbers can. Instead of units, tens and hundreds, you have units, twos, fours, eights, sixteens, thirty-twos and so on. The computer thinks in binary.
BIT	A binary digit (Ø or 1). The smallest unit of information recognizable to the computer. Can also be represented as high/low voltage (as inside the computer) or positive/negative magnetic pulses, as on cassette.
BRANCH	A point in a program where consecutive line-number execution is halted and the computer runs another part of the program—a function or subroutine, for instance.

BREAK	To stop a program in the middle, usually with the result that RUN has to be typed in to start the program again, or CONTINUE to get it to resume where it left off.
BUG	A mistake in a program. Sometimes easy to spot—particularly if it is something like a misspelled Basic word. At other times bugs are not so easy to spot—or at least the results are easy to spot but the causes in the program are difficult to find.
BYTE	Eight bits. Considered as a unit inside the computer. Memory is measured in bytes or kilobytes (1Ø24 bytes, also called k) or megabytes (1Ø24k bytes). It normally takes one byte to store one character.
CALL	A branch within a program to a subprogram or subroutine. User-defined functions can be used within BASIC programs on the Enterprise using the keyword CALL.
CARTRIDGE	A plastic box holding one or two chips which contain a program. It can be plugged into the slot at the side of the Enterprise, providing a program which will run instantly without any loading or typing. Video consoles use cartridges to play games. You just swap the cartridge when you want a different game.
CHANNEL	A 'route' through which information can enter or leave the computer, or pass between different parts of the computer. Normally the use of a channel can be re-defined by a program.
CHARACTER	A symbol used to represent information—letters, numbers, operator signs and punctuation marks are all characters.
CHIP	A little box of microscopic circuits inside the computer. A chip itself is about 2mm square (though it can vary in size), made of silicon which is known as a semi-conductor or metalloid. It is packaged in a celluloid case to protect the delicate circuits, and has metal pins to make connection with other chips inside the computer. Silicon chips are also known as Integrated Circuits.
COAXIAL CABLE	The cable which connects the computer to the TV.

	Technically, a cable where the central core is completely surrounded by another conducting layer, to prevent interference.
CODE	See CONTROL CODE. 'Code' also describes the text of a program (source code) and the program as run by the computer (object code). See also MACHINE CODE.
CODING	A description for the technical process of writing program lines, as opposed to the design of programs.
COMMAND	A program keyword, eg GOTO, CALL, etc, which gives the computer direct instructions. The first keyword in a program line is usually a command.
COMPILER	A special program which translates from one computer language to another — normally taking source code in a high-level and producing machine object code. Unlike an INTERPRETER, you cannot make instant changes to a program which uses a compiler, but generally the program runs faster when it has been compiled.
CONCATENATION	Joining two strings together using the & sign. A$ & B$ would be one long string consisting of the contents of A$ followed by the contents of B$.
CONCURRENT	Something occurring at the same time as something else. Your breathing is concurrent to your heartbeat.
CONDITIONAL	Describes part of a program which uses conditions to make decisions. IF/THEN is a conditional statement; it could loosely be translated as: 'On the condition A > 1Ø THEN...'
CONSTANT	A number or string which does not change. For example, the word PI signifies a constant (3.14...), used in calculations on circles. 2 is a constant; "A" is a constant. There are cases when what is known as a 'variable' is in fact used as a constant throughout a program.
CONTROL CODE	A character which is not visible on the screen, but which instead causes some action on the part of the computer. Examples of these are CHR$(8), which is 'backspace,' and CHR$(13) which is 'carriage return'.

CO-ORDINATE	A number which specifies the position of something, especially on the screen. PRINT AT uses two co-ordinates, vertical and horizontal, to specify where to print a character. PLOT also uses co-ordinates to specify the positions of dots or the beginning or end of a graphic line.
CPU	Central Processing Unit. A big chip inside the computer which controls all the other chips. Assembler Language is written to 'talk' directly to the CPU and varies according to the type of CPU. CPUs are also called processors. The Enterprise uses a Z80 CPU.
CRASH	Dramatic failure (either of a program or of the computer, to work)—for a variety of reasons.
CTRL	An abbreviation for 'control'. Used as a key to generate control codes directly from the keyboard.
DATA	(1) A statement in BASIC to tell the computer you are including constant words or numbers for use in a program; these are brought into use with the READ command. (2) Letters or numbers which are used by a program to make up useful information or to perform a task. The numbers used in defining your own characters are data.
DATABASE	A store of data. Often also used as a shorthand description of a program designed purely to store and manipulate data—postal addresses, for instance. Databases range in complexity from the kind which allows you to do no more than type in information and get it back as you typed it, to very complex programs which allow you to design a whole filing system on computer and/or perform sorting or extraction operations.
DEBUG	To search for and then correct mistakes in a program. This is a crucial stage in the development of big or complex programs. Mistakes are not always obvious, and any program should be carefully tested for all manner of possibilities.
DECIMAL	The counting system to which we are all accustomed, using a base of 1Ø.

DECLARATION	Telling the computer you are going to use a particular variable or array.
DEFINE	To specify, especially for later use, the details of something. Usually applies either to a character designed by you, or to a function.
DEVICE	A machine, for instance the TV or a cassette recorder, connected with the computer and run by it in some way. The sound generator and graphics are both thought of as devices by the Enterprise. Any peripherals attached to the Enterprise are treated as additional devices which are controlled by BASIC via the operating system.
DIMENSIONS	Used when describing arrays. ARRAY(X) is a one-dimensional array, ARRAY(X,Y) is two-dimensional.
DISK	A round slip of magnetic recording material (similar to tape), used in conjunction with a disk drive to store programs and data which can later be read back again. Much faster method of storage and return than cassette. Disks come in a permanent plastic sleeve, but if removed from the sleeve (only to be done to disks which are irreparably damaged in some way!) they look rather like a flexible record. The normal disks for attachment to the Enterprise are 3½ inch 'microfloppies'.
DISPLAY	Anything which appears on the TV screen.
ELEMENT	One part of an array. The number of these in an array is specified by you when you declare it.
ESC	Short for 'escape'. Used either to move to an outer program level or to indicate that following codes have a special meaning.
EXECUTE	To carry out, either a command or a program.
EXPONENT	The power to which a base is raised. 1Ø^3 would be 1Ø*1Ø*1Ø, that is 1Ø cubed. 3 is the exponent. See INVOLUTION.
EXPRESSION	A group of numbers or words which will have a value when it has been calculated. A numeric expression in

	BASIC could be something like: X*6. BASIC also has string expressions.
FIELD	Part of a record in a file. For instance, a postal address might be a record. The person's name would be one field, the street name another and so on. Fields allow for more detailed manipulation of information by making parts of it easier to find.
FILE	An organized collection of data. Can be a program file or a file of data to be used by a program. Stored on disk or cassette (or listed onto a printer). A file is divided into records.
FIRMWARE	A program that is always in the computer. The EXOS operating system in the Enterprise is firmware. The BASIC on the Enterprise is on a removable cartridge. On other home computers this program, called an INTERPRETER, could also be firmware.
FUNCTION	A part of a program which does one specific task or calculation. It is regarded as a 'program within a program'. The computer only uses it as and when instructed. A function can be either predefined—provided as part of BASIC (eg INT, CHR$, LOG)- or it can be put together by you. The eight special programmable keys on the Enterprise are known as function keys because you can redefine the function they perform.
GRAPHICS	The capability of a computer to make pictures or use lines and special symbols to set out information.
HARDWARE	The physical part of a computer system. The machinery, whether it be part of the computer itself or a separate, but connected, device.
HEXADECIMAL	A counting system using a base of 16 instead of 1Ø. Hex (for short) uses the digits Ø-9 and A-F (F is 15). Hex numbers are sometimes prefixed with & or ended in H to show they are Hex (26H is 38 decimal). Hex is used as a convenient way to represent binary numbers.
IDENTIFIER	Name given to identify a variable, function, device or any other component of the computer or program. Use of meaningful identifiers makes programs easier to

	understand.
INFORMATION	This is sometimes thought of as the result of processing data. This means the results of sorting or searching through files and/or fields. Information is data assembled into a meaningful form. Information is also anything handled by a program, ie numbers, strings, variables.
INPUT	Any data or program which goes into the computer, either from a storage device (disk etc) or through the keyboard.
I/O	Input/Output. Used to refer to any part of the computer which deals with the flow of data to and from it.
INTEGRATED CIRCUIT	Another word for silicon chip.
INTELLIGENT SOFTWARE	Software which performs intelligently. Intelligent Software Ltd are the creators of the Enterprise, among other things.
INTERFACE	A piece of hardware which forms a link between the computer and something outside it. The cassette and TV connectors are linked to interfaces inside the Enterprise.
INTERPRETER	A special computer program which interprets from one computer language to another. The BASIC on the Enterprise is an interpreter. Interpreters are different from programs which make a once-and-for-all translation between two computer languages. These are known as COMPILERS.
INVOLUTION	Raising a number to a power, eg 2^3; see EXPONENT.
JUMP	The same as BRANCH.
KEYWORD	Any BASIC word. Each one has a significance of its own.
KILOBYTE	1Ø24 bytes (the nearest round binary number to 1ØØØ).
LANGUAGE	A means of programming a computer (a system of communication with it). BASIC is a language. Other programming languages include Pascal, Fortran,

	Forth, Lisp and Logo. Each one is designed for a specific type of programming.
LOAD	To bring something out of storage and into the computer's memory.
LOGIC	The science of reasoning. Computers are designed to follow the rules of logical decision-making. The keywords AND and OR are logical ones.
LOOP	Repetition within a program by redirection back to a line which is the beginning of the loop. A part of a program the end of which leads back to the beginning unless an exit condition is fulfilled. An infinite loop is a loop with no exit condition, and is the normal cause for a computer to 'hang up' and do nothing useful.
MACHINE CODE	Real machine code is all in Binary. It's how the computer really works, and what you use—BASIC—is a program run by the machine code instructions in the computer. Assembler language describes machine code directly, using letters and symbols to represent the binary numbers.
MAINFRAME	A really big computer. Some of them are big enough to fill several rooms.
MEGABYTE	Approximately a million bytes. Equal to 1Ø24 kilobytes.
MICROSOFT BASIC	A generic description for BASIC interpreters designed by Microsoft, Inc. in the USA. Most of the features of Microsoft BASIC can be used as a sub-set of the features provided on the Enterprise's BASIC.
MODULAR	Made up of smaller, interconnected parts.
NETWORK	A group of computers, all linked together and able to communicate among themselves. The Enterprise uses a scheme known as the Intelligent Net.
NULL	Empty or with a value of zero. A null string has no characters in it. A null character has a code value of zero.
NUMBER CRUNCHING	A slang expression which means very fast and very complex calculations on numbers. Some computers

	are specially designed as number crunchers—they are fast and adept at performing large sequences of calculations quickly.
OPERAND	A number or variable which is the subject of an operation. In the expression 6*2, 6 and 2 would be operands. Also sometimes known as ARGUMENTS.
OPERATING SYSTEM	A program which controls all input and output. Mostly used with disk systems to control the running of the disk and the organization of data on the disk. CP/M is an operating system. The Enterprise contains a very extensive operating system to control all the hardware features of the machine and devices which may be attached.
OPERATOR	A mathematical word used to refer to the characters which signify operations, for example *,/ , + and =, which signify multiplication, division, addition and equality, all of which are mathematical operations.
OUTPUT	Anything leaving the computer; displays, sounds, printed listings, results.
PARAMETER	A variable which is given to a function or assigned a value. In TAN(X), X is the parameter (or ARGUMENT).
PASS	A single execution of a loop. Also used as a verb to describe the transfer of variables or values between two programs, or two parts of the same program.
PERIPHERAL	A word used to describe any machine which works under the control of a computer—TV, cassette, printer, for instance. A device used by a computer but not part of it.
PIXEL	A single dot on the TV display.
PORT	A connecting socket which links the computer (through an interface) to a peripheral device.
PORTABILITY	The quality (quite rare) in a program of being usable on more than one system.
PROCEDURE	Another name for sub-program. With the BASIC on the Enterprise, functions provide all the normal features of

	procedures.
PROCESSING	Carrying out operations on data. This includes calculating with numbers, graphics (using co-ordinates etc) and, in fact, everything a computer does.
PROGRAM	A series of ordered instructions which set out a task for the computer to do.
PSG	An acronym for Programmable Sound Generator. A chip inside the Enterprise which sends signals to a loudspeaker, thus producing sound.
RAM	Random Access Memory. Memory which is used for the temporary storage of programs or data. It empties itself when the word NEW is typed or when the computer is switched off.
READ	To take something out of store and bring it into current use. Can be synonymous with LOAD, although it has a slightly different meaning when used as a BASIC command.
REAL TIME	Time which bears a relationship to the real world, instead of being related to the inner workings of the computer. Often used when describing the processing of data simultaneously as it is input. INKEY$ is almost a real-time operation. Arcade games work in real time, to give the impression of real things happening very fast. Real time is only provided on certain computers or in certain programs.
REFERENCE	Normally used when describing a parameter which is given to a function and actually contains a variable rather than simply the current value of the variable.
RESERVED WORD	A BASIC word which cannot be used in BASIC for any other purpose. Most BASIC words can be used as variable names. Those which cannot are reserved words.
RESOLUTION	The number of dots available for plotting on a screen. This is governed by the quality of the graphics available on the computer. Resolution can be high or very low. The higher it is, the better the definition of lines and shapes when they appear on the screen. The

	highest resolution on the Enterprise allows 672 dots horizontally and 512 dots vertically on the display.
ROBUST	A description applied to a program or piece of software which has very few bugs in it, and which works well whatever users do with it.
ROM	Read Only Memory. Memory which contains firmware. It cannot be changed by you, although it can be used or looked into. Remains the same when the computer is switched off.
SEARCH	To look through a file or list for something in particular.
SOFTWARE	A program is software. Instructions to the computer which can be changed.
SORT	To reshuffle a file or list into a particular order, eg alphabetical.
SPIKE	A rapid surge or dip in mains voltage. Can cause very occasional problems with disk systems and can temporarily switch off a computer, losing any program which is currently working (this does not happen often).
STATEMENT	A self-contained, meaningful part of a program. IF/THEN lines form statements. IF A=2 on its own would not have much meaning. IF A=2 THEN PRINT "A=2" would.
STORE	As a verb, to put information or data away for later use, either temporarily in RAM or permanently on cassette or disk. Memory or peripherals on which you can save information are sometimes called storage or mass-storage devices. Store is also used as a noun, to describe memory which can be written to (RAM or disks, for example).
STRING	A sequence of characters which are not understood but can still be processed by the computer. In BASIC, within a program listing, a string appears either as a variable with $ on the end or as a series of characters in inverted commas.
STRUCTURE	The organization of parts of a program; as distinct from an algorithm, which is a plan of how a program will

	solve its problem.
SUBROUTINE	Very similar to a function. A part of a program which does one specific task under the control of the rest of the program. A function has a specific beginning and end. A subroutine is controlled with the words GOSUB and RETURN, on the basis of line numbers. A function is given a name.
SUBSCRIPT	A reference number given when accessing an array. For example, with ARRAY(X), X is the subscript.
TEST	A tryout of a program to see if it works. Or, in programming terms, 'taking a look' at something to see if it fulfils a condition. The computer tests variables in IF or CASE statements to see if they fit in with any of the statements.
UNIT	Can either mean one of something—a character is a unit of data—or a device. The computer is a unit. A disk drive is a unit.
USER	The person using a computer or program. User-friendliness refers to the ease with which some programs may be used.
VARIABLE	A number or string which may change, and is therefore given a name by which you and the computer will recognise it.
WIPE	To empty, either a storage medium (disk or cassette, for instance), or the computer's memory.
WRITE	To put data or information into. You can write data into a file; for instance.

INDEX

Design by David Pearce/Liz James
Typesetting by Accent Design and Print

CONTINUE
START
FUNCTION 1
REMEMBER
LIST
FUNCTION 2
COPY
ECHO
FUNCTION 3
REMOTE 2
REMOTE 1
FUNCTION 4
MUTE
CLICK
FUNCTION 5
ESC
TAB
CTRL
LOCK