Chapter Two
BBC BASIC

Not only is the BBC Micro a remarkable and interesting machine from the hardware point of view, it also has some equally impressive software. One of the interesting characteristics of the software is the way that it interacts with the hardware to produce something that is extremely versatile. For example, the sound generator chip is fairly sophisticated in that it has three tone channels and a noise channel, but when you add in the ENVELOPE command it behave's in a way that seems to exceed its specification! Much of the way that the sound generator appears to the user is entirely the invention of well-conceived software. In this chapter we look at BBC BASIC which forms roughly half of the resident software in the machine. (The other half, the MOS, is considered in Chapter Three.)

Rather than going through a point-by-point discussion of the commands that make up the BASIC, a task already accomplished by the User Guide, the first part of this chapter looks at some of the features and commands that either make BBC BASIC special or are in some way difficult or unusual. In the second half of the chapter we take a look inside and find out how the BASIC interpreter runs and stores your BASIC programs. This sort of information is often interesting and is worth knowing for its own sake. Also, there are many practical reasons for delving into the interpreter. Knowing how the BASIC is implemented can suggest the fastest and most economical ways of doing things. It can also suggest 'short cuts' and ways of accomplishing what would otherwise be impossible (for example, printing a list of all the variables in use). Finally, the assembly language programmer needs to know something of how variables are stored in order to make use of the parameter-passing capabilities of the CALL statement (see Chapters Seven and Eight).

BBC BASIC, a BASIC with structure

When Acorn were approached by the BBC to produce a computer, one of the specifications was that its BASIC should include statements not normally thought of as being part of the language. The reason for this was the desire to make BASIC a more academically respectable language and to make it able to take advantage of the method of programming known as 'structured programming'. The theory behind this method of programming is not within the scope of this book but its practical interpretation has come to mean the use of the commands:

IF . . . THEN . . . ELSE
REPEAT . . . UNTIL
WHILE . . . DO . . .
and
FOR . . = . . TO . .

BBC BASIC doesn't include all of these statements; it lacks WHILE, but even so it can claim to be a 'structured BASIC'. If you want to treat BBC BASIC like an ordinary BASIC, that is programming using only IF ... THEN, GOTO and FOR, then you can. However, if you want to write BASIC in a structured way there is more to it than just adding the more complete form of the IF and the REPEAT... UNTIL statement to your repertoire. It is the aim of structured programming to produce programs that are easy to undertstand and as bug-free as possible and this can be achieved in many ways. The most important thing is to try and make the 'flow of control' through your program as simple and obvious as possible. You can do this by restricting the way that you use GOTO, by using REPEAT ... UNTIL and FOR to form loops in preference to GOTO and by using subroutines to group statements together into logical units. To go into any more detail about structured programming would take us far from our subject. If this brief introduction has whet your appetite then you can find out more about structured programming and good programming style in general from my other book, The Complete Programmer. However, it is worth looking at the subject of subroutines a little more.

Subroutines, procedures and functions

One of the biggest criticisms of BASIC is that it has a very limited

ability to group statements together into logical units. True, you can use GOSUB and RETURN to form subroutines but this has a number of shortcomings, as follows:

1. You have to refer to subroutines by a line number rather than a name that indicates the subroutine's purpose.
2. Subroutines and the rest of the program have unrestricted access to each other's variables.
3. There is no way to isolate the variables that supply inputs to and return results from a subroutine - in other words there are no facilities for parameters.

The first problem can be overcome to a certain extent by assigning the line number to an appropriately named variable. For example, if you wanted to call a subroutine starting at line 2000 that sorted an array into order, instead of:

```
GOSUB 2000
```

you could use

```
SORT = 2000
GOSUB SORT
```

However, this technique makes renumbering a program very difficult. You may not see why the second point is a problem. Why shouldn't a program and a subroutine share the same variables? There are a number of valid arguments to suggest they should not. As subroutines should be collections of statements that carry out identifiable tasks, they are often written without reference to the rest of the program. For example, a subroutine that sorts an array into order is generally useful and might find its way into a number of programs. When writing such a subroutine, what names should you give to the variables that you use so that they don't clash with variables used for other purposes in the main program and other subroutines? Variables clashing in this way is a very common cause of BASIC programs failing to work as expected. If you do have a way of isolating the variables in a subroutine from the rest of the program then the third problem comes into play. If subroutines cannot have access to the variables in the main program how do they receive their input values and return their results? The solution to this problem is, of course, to use parameters in the same way as in a standard BASIC function. In fact, in many respects, the standard BASIC function would be superior to the BASIC subroutine if it were extended to allow more than one statement. This is indeed the

direction that BBC BASIC takes in improving the subroutine.

A user-defined function in BBC BASIC can take the 'one-line' form found in most other versions of BASIC. For example.

```
DEF FNsum(a,b,c) = a + b + c
```

is a function that adds three numbers together. The three parameters a,b and c are not variables in the usual sense of the word in that they do not 'appear' in the main program because they have been used in the function. There is even no problem if the main program uses variables with the same names. In some senses the names used for parameters are only relevant within the function definition. In other words, parameters are *local* to the function. Another thing to notice is that a function returns a single value as a result. It is this that makes it possible to use functions within arithmetic expressions. From the point of view of evaluating expressions, a function behaves just like an ordinary variable except that the value associated with its name is produced by the function rather than just stored in memory. However, even though this is a very useful feature there are many occasions when a subroutine needs to return more than a single result.

BBC BASIC improves the standard BASIC function to allow multiple statements in its definition. For example, it is possible to write a function that finds the larger of two numbers:

```
100 DEF FNmax(a,b)
110 ang=a
120 IF a<b THEN ans=b
130 =ans
```

Line 100 states that what follows is the definition of a function called FNmax. Lines 110 and 120 place the larger of a and b and place the result in ans. Perhaps the oddest looking line is 130. Multi-line functions always end with an assignment with no variable on the left-hand side. The effect of this is to set the value returned by the function to the result of the expression on the right of the equals sign, Once the value of the function has been determined it is used in the evaluation of the expression in which the function occurred in the main program. Once again it is useful to think of this last assignment statement as *storing* the result of the function in a *dummy* variable with the same name as the function. The parameters a and b are, again, local to the function. But, what about the variable ans? This is declared within the function so you might expect it to 'belong' to the function. In fact it is a

variable that is part of the rest of the program. The variable ans is no different from any other variable in a standard BASIC program. This is something of a disadvantage if there is already a variable ans in use in the main program. If this is the case and FNmax is used then the variable will change its value even if its use in the main program has nothing to do with finding the maximum of two numbers. Such changes caused by functions in *innocent* variables in the main program are called the *side effects* of the function. If you want to write programs that are not only easy to understand but easy to debug then the functions used in your programs shouldn' tcause any side effects. BBC BASIC provides the LOCAL statement for just this reason. A variable that is declared in a LOCAL statement behaves much like a parameter in that it has nothing to do with any variables of the same name used in the rest of the program. For example, the FNmax function can be written as:

```
100 DEF FNmax(a,b)
110 LOCAL ans
120 ans=a
130 IF a<b THEN ans=b
140 =ans
```

Now the variable ans is declared as local by line 110 and, just as the names a and b have nothing to do with the main program, the value of ans may be changed without affecting any variable in the main program. This version of FNmax has no side effects! We will consider how parameters and local variables actually work later on in this chapter. However, it is worth pointing out that if you use a parameter or declare a local variabiethatdoesnothaveacounterpart with the same name in the main program then one comes into being as soon as the function is used. (Numeric variables are given the initial value zero and string variables are set to the null string.)

Functions are very useful but they do have the drawback that they can only return one result (excluding side effects). BBC BASIC provides an additional feature - the *procedure* - that is supposed to get around this problem, If you want to return anything other than a single result you should use a procedure. Procedures are a very useful feature of BBC BASIC but the one thing they do not solve is the problem of returning more than one result! In fact there is no clean way of getting any results back from a procedure. A procedure is defined in much the same way as a function but it ends with ENDPROC. For example,

```
10 DEF PROCmaxmin(a,b)
20 IF a>b THEN max=a:min=b ELSE max=b:min=a
30 ENDPROC
```

Line 10 defines the procedure maxmin with two parameters a and b. Line 20 does all the work by placing the two results correctly in the variables max and min. The final line simply marks the end of the procedure in the same way that RETURN ends a subroutine. The two parameters are local in the same way that the parameters in a function are. However, the only way that the two results can be communicated back to the main program is by the use of two non-local variables max and min. As these are non-local, any variables of the same name in the main program will be altered by the use of the procedure. In this sense the only way that a procedure can return any results is by making use of side effects! Any variables used in a procedure that are used purely for internal purposes should be named in a LOCAL statement in an attempt to minimise unnecessary side effects but, unless no results are to be returned, procedures must produce side effects.

Even though procedures have this fundamental shortcoming they are extremely useful - so much so that they are always to be preferred to the standard BASIC subroutine. Although there is much written in the User Guide it is worth highlighting a number of important points concerning functions and procedures:

1. Use a function whenever a single result is to be produced.
2. Place *all* the variables used in a function in a LOCAL statement to remove all side effects.
3. All input to functions and procedures should be via parameters whenever possible.
4. Use a procedure if no results or more than one result is to be produced.
5. In a procedure, name all variables not used to return results in a LOCAL statement to remove unnecessary side effects.
6. Parameters can be any of the three simple variables - real, integer or string - but not an array. Arrays can only be passed as non-local variables.
7. Functions can return strings as results.
8. Functions and procedures are always to be preferred to BASIC subroutines and should be used as often as possible. One good reason is that procedure calls are faster than subroutine calls.
9. The variables used in a procedure or function are only created

after the function or procedure is first used.

10. Functions and procedures work slightly faster the second time they are used.
11. Functions and procedures can be called recursively.

The reasoning behind many of these points will become clear as the chapter progresses. However, the final point is worth illustrating with an example. Whenever a function or procedure is called it creates a completely new set of local variables. This fact means that a function or procedure can call itself, i.e. can be used recursively. Recursion is a subject that is dealt with extensively in many an academic textbook. Most people find it difficult to cope with and it is therefore fortunate that it is rarely actually needed in the solution of practical problems. As an example of recursion consider the problem of writing a function to calculate n! (n factorial). The usual way to write a function that calculates n! is by using a FOR loop. (n! is the product of all the integers from 1 to n, that is:

n*(n-l)*(n-2)*. *1

from 1 to n).

```
100 DEF FNF(N)
110 LOCAL I,SUM
120 SUM=1
130 FOR I=1 TO N
140 SUM=SUM*I
150 NEXT I
160 =SUM
```

The FOR loop at lines 130 to 150 calculates a running product in SUM that is equal to N!.

There is another way to approach the problem of calculating factorials. If you want to know what n! is you could find out by calculating (n-l)! and multiplying it by n. In other words, n!=n*(n-1)!. For example, 4!=4*3!=4*3*2!=4*3*2*1! and we know that 1! is 1. This idea results in the following function:

```
100 DEF FNF(N)
110 IF N<>1 THEN =N*FNF(N-1)
120 =1
```

Line 110 looks very strange in that the expression to the right of the equals sign uses the function FNF. To see how this works follow through the calculation of FNF(3). When FNF is first called, the value

of N is 3 so the expression following line 110 is carried out. This involves calling FNF once more but with the value of the parameter N equal to 2. Remember that when FNF is called for the second time a completely new set of variables is created. This second call to FNF also results in FNF being called again but this time with a parameter value of 1, which causes this third call to FNF to finish via line 120. This results in the value I being returned as the result of the third call which allows the evaluation of the expression in the second call to be completed i.e. 2*1 and the result passed back to the first call to FNF. Finally, this result allows the expression in the first call to FNF to be completed giving the correct answer 3*2*1.

If this description has left you feeling confused then you are not alone! It is possible to follow the execution of the function through all its ' incarnationsbut you really need pencil and paper to do it easily. Recursion is something that you either feel comfortable with or find difficult. You can write recursive functions and procedures in BBC BASIC. However, because procedures do not return results except via non-local variables they are much more limited in the way they can be used recursively.

Indirection and hexadecimal

The provision of REPEAT . . . UNTIL functions and procedures certainly make BBC BASIC a ' higherlevel language than standard BASIC. However, there are one or two extra facilities included in BBC BASIC that makes it easier to use for lower level tasks. In particular, there are three ' indirectionoperators that make the direct manipulation of memory easy.

Most versions of BASIC provide the POKE command to alter a memory location and the PEEK function to examine the contents of a memory location. BBC BASIC replaces both of these facilities by a single indirection operator ' ?'Writing a question mark in front of a number or a variable causes it to be interpreted as the address of a single memory location. So, for example, ?40 is a reference to memory location 40. To find out what is in a memory location all you have to do is use PRINT ?address. To change the contents of a memory location you simply write ?address=new value. If you read the question mark as ' thememory location whose address is' then you should be able to understand any use of the indirection operator.

Although it is useful to be able to handle single memory locations the BBC Micro tends to work with more than one location at a time. For example, it uses four memory locations to store an integer. To make the manipulation of multiple memory locations easier two other indirection operators, ! and $ are available. The exclamation mark works in the same way as the question mark but it refers to four memory locations. To be exact the statement !address refers to the four memory locations whose addresses are address, address+1, address+2 and address+3. These four locations are treated as if they were a standard integer value (with the most significant byte stored in address+3). The dollar sign $ is a little more difficult to understand in that the number of memory locations that it refers to is variable. It is known as the *string indirection* operator, because it deals with memory in terms of strings of characters. For example, $4000="ABCD" stores the ASCII codes for A in memory location 4000, the code for B in 4001 and so on until it stores the code for D in memory location 4003. To mark the end of the string it then stores the ASCII code for carriage return in 4004. In the same way PRINT $4000 will print the character corresponding to the ASCII codes stored in the memory locations starting at 4000 and going on up to the first occurrence of the ASCII code for carriage return.

You can specify an *offset* with any of the three indirection operators. For example, 4000?10 refers to memory location 4010. In other words, address?offset is the same as ?(address+offset). This is a useful facility for *stepping* through a range of addresses but it can be confusing for the beginner.

The indirection operators certainly provide a way of handling memory locations but both memory addresses and memory contents are usually specified in terms of hexadecimal rather than decimal. The main reasons for this are that you can specify a memory address using four hex digits and the contents of a memory location using only two hex digits. BBC BASIC is capable of handling numbers written in hex and printing values in hex. Writing ' &ln front of a constant indicates that it should be taken to be a hexadecimal number. For example, &F is 15 and OFF is 255. In particular, it is important not to confuse &l0 with ten (to find out what it is type PRINT 4 10). The use of & may confuse many people because $ is the most common symbol for hexadecimal but you soon get used to it. To print a number in hex all you have to do is place in ~ front of it. For example, PRINT~255 produces FF. You can use both & and ~ in combination with any of the

indirection operators to manipulate memory directly in hex. This is a very useful facility as we shall see in this and later chapters.

BASIC's use of memory

The memory map given in Chapter One showed the general layout of the BBC Micro' saddress space in terms of ROM, RAM or I/0. However, when running BASIC, the available RAM has a fairly fixed use that can be seen in Figure 2.1. The top-most portion of RAM is always taken by high resolution graphics. The actual amount that is

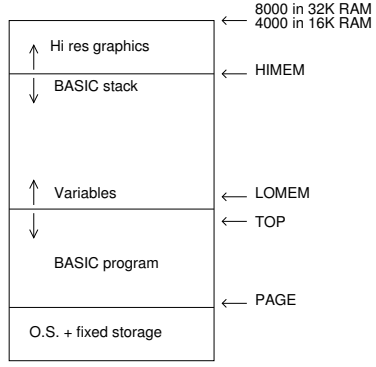

Fig. 2.1. RAM as used by BASIC.

used depends on the graphics mode selected (see Chapter Four) but the address of the first memory location below the area used for graphics is always available in the variable HIMEM. The bottom-most portion of memory is also used for something other than storing BASIC programs. It is used by the MOS (see the next chapter) to store details of how the machine is set up, i.e. what type of printer is in use etc., as storage for buffers such as the keyboard buffer and sound buffer etc., and as RAM storage for any programs in ROM. All-in-all, the lower area of memory is used foi just about everything! An important area from the point of view of BASIC is &0400 to &0800 which is designated as the *language work area*. This is used by the BASIC ROM to store most but not all, of the information about a program as it is running. The area of memory from &0000 to 8<0 1 00 is known as page zero and it is particularly useful because certain machine code instructions (see Chapters Seven and Eight) only work with page zero. Thus, even though BASIC has a work area set aside for it, it does use

some zero page locations. The actual amount of memory used by the MOS and other ROM programs varies according to what the machine configuration is. However, the address of the first free memory location can always be found in the variable PAGE. As a BASIC program is typed in or loaded from tape it is stored in memory starting at? PAGE. The ' top'of the BASIC program - in fact the first free memory location after the BASIC program - can be found by using the function TOP. The variable LOMEM contains the address of the first memory location above the BASIC program that can be used to store variables that are created when the program is actually run. LOMEM usually contains the same address as TOP but you can change LOMEM to point to another area of memory if you want to (reasons why you might want to are suggested in Chapter Eight). The amount of memory that a BASIC program is using, not including any memory needed for variable storage, can be found by typing PRINT TOP-PAGE.

It is interesting to go through the changes in memory use that occur as a program is loaded and run. When the machine is first switched on all the memory pointers are set to their correct values. As a BASIC program is typed in or loaded, the value of TOP and LOMEM are adjusted so that they always point to the first free location above the program. When the program is RUN, memory is used starting from LOMEM to store variables as they are encountered within the program.

The area of memory just below HIMEM is also used for storage by a running BASIC program but only for temporary storage for things such as the return address for subroutines and procedures etc. Thus, as the program runs, memory is taken starting at LOMEM and extending upward - this is often referred to as the BASIC heap - and growing downward from HIMEM, which is often referred to as the BASIC stack. Obviously, if the running program changes the display mode then the value of HIMEM will change. If this were to happen when the BASIC stack was in use then it would crash the program; this is the reason why you can only change modes from the main program and not in a subroutine or procedure.

Now that we have a picture of how BASIC puts the RAM to work it is time to examine in detail how things are stored. First we will look at how the lines of a BASIC program are stored and then at how the different types of variables are stored.

The way BASIC is stored

Each line of BASIC that you type in has three parts - the line number, the keyword such as GOTO PRINT or REM, and the rest of the statement. This division also corresponds to the way that a line of BASIC is stored internally. The ASCII code for carriage return, i.e. 820D, marks the start of every line. Then follow two bytes that hold the line number in binary. The fourth memory location is used to store the length of the line. Finally, we reach the actual text of the BASIC statement. This is stored exactly as written in the form of ASCII code but with a few changes. For example, any keywords in the line are replaced by codes that can be stored in a single memory location. This makes good sense because it saves storage space and the code that is used is related to the ROM address of the machine code that implements the BASIC operation. (For a full list of key words see the User Guide.) This changing of keywords into codes is known as *tokenisation* and the codes are known as *tokens*.

You can see the format of the internal storage of a BASIC line in Figure 2.2. When there is no program in memory there is still a single

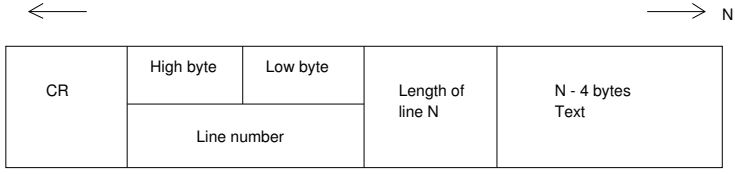

Fig. 2.2. Internal storage of a BASIC line.

carriage return stored so PAGE and TOP never point to exactly the same location. The end of a BASIC program is marked by a line number that has &F stored in the high byte of the line number. BBC BASIC line numbers must lie in the range 0 to &7FFF so using &FFxx as an end of program marker doesn' tinterfere with the normal numbering of a program. 1f you type NEW then all the BBC Micro does to delete the program is to write &FF in the high byte of the first line number and reset the pointers TOP and LOMEM. Because deleting a program with NEW doesn' talter anything else about the program, it is possible for OLD to restore the program by setting what was the high byte of the line number of the first line back to zero and then use the length of line information to scan through the memory to find the original end of the program and set the pointers TOP and LOMEM accordingly.

An interesting point is that neither LIST nor SAVE actually take any notice of TOP to find the end of a program. Instead they both use the end of program marker in the high byte of the line number to stop listing or saving a program. The sole purpose of TOP seems to be to govern where a new line of BASIC would be added to the program. It is also worth pointing out the amount of work that is involved in obeying a simple GOTO or GOSUB command. The program has to be searched for the line number used in the GOTO or GOSUB. This involves starting at the beginning of the program and comparing line numbers one at a time, using the length of line information to move on to the next line number until either the search is successful or the current line number is bigger than the one being searched for. This long-winded process is, of course, the reason why functions and procedures are faster than subroutines.

Handling variables - format and storage

There are two things of interest about the way BASIC handles variables. First, it would be interesting to know how to find out where any variable was stored. Secondly, it would then be useful to know the format used to store information in the different types of variable.

BBC BASIC uses a very clever method of keeping track of where it has placed a variable. When a program runs, each new variable that is encountered is allocated some space in the BASIC heap (starting at LOMEM). The address of the first free memory location in the heap is stored in &0002 and &0003 which, for want of a better name we will call ' freemem'Thus, the storage of variables starts at LOMEM and goes up to freemem, New variables are allocated memory where freemem is pointing and then freemem is increased to point to the next free location. So far this is exactly what any other version of BASIC does to allocate storage to variables. What is special about BBC BASIC is the way that it keeps track of where each variable is.

Other versions of BASIC simply store the name of each variable as it occurs along with its value. When a variable is required a search is carried out of the entire BASIC heap. If you use a lot of variables this can take a long time! BBC BASIC, in an attempt to shorten the search time, keeps a separate list of variables for each letter of the alphabet, upper and lower case. In other words, if the first letter of a variable' s name is A, it joins the ' capitalA' list. If its first letter is a z, then it joins

the lower case z list. When BBC BASIC wants to find a variable, it has only to search the list of variables that have the same first letter. As long as you don't start all your variables with the same letter this should be a quicker way of finding them.

The way that the separate lists are maintained is fairly simple. For each letter of the alphabet A - Z and a - z there is a start of list pointer which contains the address of the start of the list of variables that start with that letter. These pointers are stored in the language work-space area of memory from &0482 and &0483, which forms the pointer to all the variables starting with A, to &04F4 and &04F5, which points to the start of the list of variables beginning with z. To work out the address of the pointer to the list of all variables beginning with the letter stored in A$, use (ASC(A$)-65)*2+&0482 which gives the address of the least significant byte of the pointer. If there are no variables beginning with a particular letter then the corresponding pointer is set to zero. If there are variables beginning with a particular letter then the corresponding pointer contains the address of the first variable. The address of the second variable in the list is stored in the first two memory locations allocated to the first variable. Each variable in the list contains a pointer to the next variable in the list. The end of the list is marked by a zero address for the next variable.

The procedure for adding a variable to the BASIC heap is now a little more complicated than for the simple storage scheme described earlier but the increase in speed that results is well worth the trouble. To add a variable to the list you first have to find the end of the list by searching down its length until you find the first zero pointer. This may sound like a chore but of course it has to be done anyway to find out if the variable already exists! When the variable at the end of the list is found its pointer is changed to point to the same location as freemem. Then the space that the variable requires is allocated and freemem is changed to point once again to the first free memory location.

Now that we know how variables are allocated space and how to find where they are stored, the only thing left to discuss is the format used to store each type of variable. BBC BASIC recognises three fundamental or 'simple' data types - integer, real and string - and can handle arrays made up of any of the three types.

The storage format used for an integer variable can be seen in Figure 2.3. The first two locations form the pointer to the next variable with the same initial letter, as discussed above. These two bytes are zero if there is no next variable. The subsequent bytes are used to

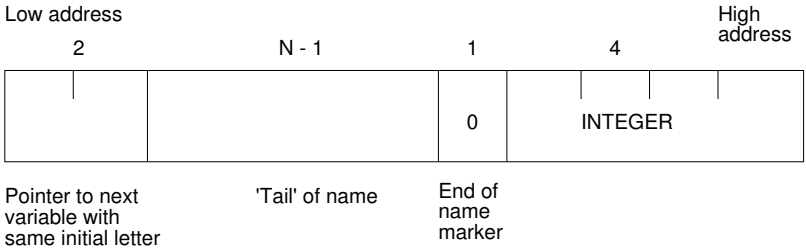

Fig. 2.3 Storage of an integer variable with N characters in its name (including the %).

record the rest of the variable' sname minus the first letter but including the % sign to show that what follows is an integer. So, for example, a variable called TOTAL% would have its name stored as OTAL%. The end of the name is marked by a memory location with zero in it. Following this are four bytes that hold the actual integer value associated with the variable. The format that is used to store the value is 64-bit 2s complement. You can use the ! indirection operator to obtain its correct value.

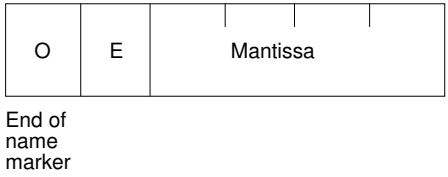

Fig. 2.4. Storage of a real variable

Figure 2.4 shows the format used to store real variables. To be more precise, it shows only the part of the format that is different from the integer format. A real variable starts off with a pointer to the next variable and the rest of its name just like an integer variable but following the end of name marker are five bytes used to store a real value. A real value is stored in floating point form with a one byte exponent and four byte mantissa.

A string variable also starts off in the same way as an integer variable with a pointer to the next variable and the rest of the name including the $ sign. The rest of the string format consists of four bytes, as shown in Figure 2.5. The first two of these four bytes contain the address of the actual string of characters that are stored in the

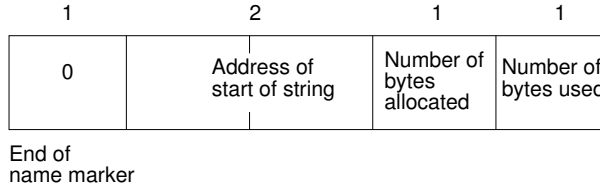

Fig. 2.5. Storage of a string variable.

string. The third byte is used to record the number of bytes allocated to the string for storing its value and the fourth byte records the number of bytes actually used (in other words, the length of the string). What is interesting about this format is that the string of characters that forms the string' svalue is stored away from the variable itself. This will be considered in more detail in the section on garbage collection.

Finally we come to the format used to store arrays. This initially follows that used for the same pattern as the simple variables but the name that is stored not only includes the $ or% sign but the (as well. The rest of its format can be seen in Figure 2,6. The first location following the end of name marker records the number of dimensions in

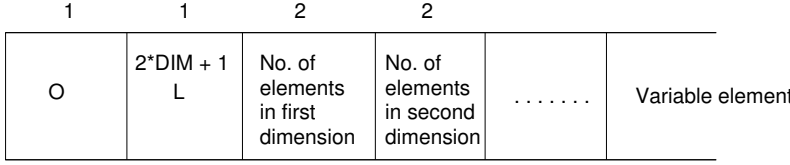

Fig. 2.6. Array storage

the array. To be precise, it is twice the number of dimensions plus one that is stored in this location which is the number of memory locations needed to store all of the other information about the array. Following this byte are pairs of memory locations, one pair for each dimension, recording the number of elements in each dimension. Following this information are the values that form the elements of the array.

Before leaving the subject of variable storage it is worth commenting on the way functions and procedures are handled. In the same way that variables are formed into lists so are functions and procedures. The memory locations &04F6 and Sc04F7 are used as a pointer to a list of procedures. Each item in the list has roughly the same format as a variable. The first pair of bytes point to the next

procedure in the list, if any. Then comes the full name of the procedure ending with the usual zero byte. Following this are two bytes containing the address of the start of the procedure. The same technique is used to form a list of functions but in this case the initial pointer is formed by locations &04F8 and &04F9.

A heap dump program

To make the above information on variable storage etc. a little more concrete a variables, procedure and function dump program is given below. Not only does this serve to illustrate the points made above but it is a useful program in its own right.

```
9000 DEF PROCdump
9010 LOCAL X,Y
9020 FOR X=&0482 TO &04F4 STEP 2
9030 Y=?X+256*X?1
9040 IF Y<>0 THEN PROCvarlist(X,Y,0)
9050 NEXT X
9060 PRINT
9070 PROCother
9080 ENDPROC

9090 DEFPROCvarlist(X,Y,P)
9100 LOCAL A$,I,TYPE
9110 A$=STRING$(50,"X")
9120 A$=CHR$(65+(X-&0482)/2)
9130 TYPE=0
9140 PRINT TAB(0);~Y;TAB(5);
9150 I=2
9160 IF Y?I=0 THEN GOTO 9230
9170 A$=A$+CHR$(Y?I)
9180 IF Y?I=ASC("Z") THEN TYPE=1
9190 IF Y?I=ASC("$") THEN TYPE=2
9200 IF Y?I=ASC("(") THEN TYPE=TYPE+100
9210 I=I+1
9220 GOTO 9160
9230 IF TYPE>99 THEN GOTO 9290
9240 IF P=1 THEN PRINT "PROC";A$:GOTO 9350
9250 IF P=2 THEN PRINT "FN";A$:GOTO 9350
9260 PRINT A$;TAB(10);TYPE;TAB(20);EVAL(A$)
9270 IF TYPE=2 THEN PRINT TAB(20);Y?(I+3);TAB(30);Y?(I+4)
9280 GOTO 9350
9290 D=(Y?(I+1))-1
9300 PRINT A$;
9310 FOR Z=1 TO D STEP 2
9320 PRINT STR$(Y?(I+1+Z)*256*Y?(I+2+Z)-1);",";
9330 NEXT Z
9340 PRINT CHR$(08);")"
9350 IF Y?1=0 THEN ENDPROC
9360 Y=?Y+256*Y?1
9370 GOTO 9120
```

```
9380 DEFPROCother
9390 LOCAL X,Y
9400 Y=?&04F6+256*?&04F7
9410 IF Y=0 THEN GOTO 9440
9420 X=0
9430 PROCvarlist(X,Y,1)
9440 Y=?&04F8+256*?&04F9
9450 IF Y=0 THEN ENDPROC
9460 X=0
9470 PROCvarlist(X,Y,2)
9480 ENDPROC
```

The first procedure, PROCdump, examines each of the variable pointers in turn and calls PROCvarlist if any variables are present in the list. Most of the work is done by PROCvarlist, which first puts together the full name of the variable - lines 9120 to 9220. While the full name of the variable is being constructed in A$, each character of the name is tested against %, $ and (to determine the TYPE of the variable. Once the full variable name is present in A$ the BASIC statement EVAL is used to print the contents of the variable by line 9260. If the variable was a string then line 9270 also prints the amount of storage allocated to the string and the amount of storage actually used. If the variable is an array then no attempt is made to print out its values; just its dimensions are printed. Line 9290 works out the number of dimensions in the array and lines 9310 to 9340 print the size of each dimension in turn. After all the details of the variable have been printed, lines 9350 to 9370 work out the address of the next variable with the same initial letter. If there is none, then control is returned to the dump procedure. After all the different variable lists have been processed, PROCother is called to print the active procedures and functions. PROCother simply checks the initial procedure and function list pointers and calls P ROCvarlist to work out the names of each procedure or function in the list.

Notice that if you use PROCdump it will not only report any variables etc. employed by a main program with which it is being used in conjunction. It will also report all of its own variables, procedures and functions.

The resident integer variables

Although we have discussed the storage and format of the variables that can be used in BBC BASIC, we have ignored a set of very special

and very useful variables - the resident integer variables. The names of the resident integer variables are @%, A% ,B% . . . Z%. Instead of being stored in the BASIC heap, these variables have a fixed area of the language work area set aside for them. As a result they exist whether you use them arnot. They are not cleared or changed in any way by NEW, CLEAR or LOAD. In fact, apart from explicit assignment, the only thing that changes the value of a resident integer variable is switching the machine off and on again! It is often useful to know that the resident integer variables are stored starting with @% at &0400 with four bytes to each variable so that A%, for example, starts at &0404. The resident integer variables will be mentioned again in Chapters Seven and Eight.

Garbage collection

When a numeric variable is allocated space in the BASIC heap it is there to stay and it never needs to change the amount of storage allocated to it. However, string variables are very different and they can change their size all the way through the execution of a BASIC program. A string may start out holding only a few characters and then grow to the maximum size a string can be, i.e. 255 characters, and then shrink back to only a few characters again. As a string grows in size, more memory in the heap has to be allocated to it. When it grows smaller it would be efficient if the memory it released were returned to the heap, and this is usually referred to as garbage collection. However, garbage collection takes time and the BBC Micro is built for speed! When a string variable is first used, an entry in the format given in Figure 2.5 is created in the heap. The actual characters that make up the string are also stored in the heap and the address of the first character and its length are stored in the appropriate locations next to the string variable' sname. In fact, a few more bytes than are necessary to store the string are allocated to allow the string to grow a bit before problems arise. If you reduce the length of the string then nothing happens apart from its current length being updated. In particular, the memory locations that are freed are not returned to the heap; instead, they are left ready for the string to increase in length again. If you add characters to the string to the point where all of its allocated space is used up, then to increase its length still more requires, obviously, some more memory to be allocated from the heap. This is not an easy matter

because other variables and strings may be located just above the string in question. Allocating extra space, therefore, would mean moving everything above the string up in the memory. Considering the way that variables are linked together in separate lists, this would be no easy operation! Instead of this difficult move, the BBC Micro simply creates a new copy of the string's value at the top of the heap including some extra memory locations for future growth. If you think about this approach to creating more space for a string value you should be able to see that it is fast but very wasteful of memory. There can be a considerable number of dead copies of string values occupying valuable RAM storage because the BBC Micro fails to do any garbage collection.

To illustrate this problem consider the following short program:

```
10 A$=""
20 PROCSIZE
30 A$=A$+"A"
40 PRINT LEN(A$),;
50 GOTO 20

100 DEF PROCSIZE
110 PRINT ?2+256*?3-TOP
120 ENDPROC
```

The procedure PROCSIZE prints the current size of the BASIC heap by working out the difference between freemem, and TOP. The program itself first sets up a string A$ that is initially set to the null string. Each time through the loop formed by lines 20 to 50 a single letter is added to the string and the size of the heap is printed. If you run this program you might be surprised how much storage it takes to hold 255 characters. The final line that the program prints indicates that the heap reaches nearly 4K bytes! The solution to this waste of storage is simple. If the string is defined to be the maximum size that it will ever be when it is first used no extra copies of it will ever be made. 1f you change line 10 of the above program to read:

```
10 A%=STRING$(255,"X"):A$=""
```

you will find that the final line of the program now reveals that it takes a much more reasonable 272 bytes to store a string 255 characters long. If you set strings equal to the maxinium length that they are likely to reach during a program you will save a lot of memory! The way that strings are handled by the BBC Micro might seem a little

crude but it really is the only way that it can be done and still achieve a fast BASIC.

LOCAL variables and the stack

Although we now know a lot about the way variables are stored we still do not know how local variables work. How can it be that a variable named in a LOCAL statement can replace any variable of the same name in the main program for the duration of the procedure or function in which it occurs and the original value stored in the variable still be intact at the end of the procedure? The answer to this question is surprisingly straightforward. When a function or a procedure is entered, any variables that are named in a LOCAL statement or that occur as parameters are searched for in the heap. If a variable with the same name is found then the value that is stored in it is stored on the BASIC stack. If the variable doesn't exist then it is created with an initial value of zero if it is numeric and the null string if it is a string. After the original value has been safely stored away on the stack, the variable can be used by the function or procedure without any worry about altering anything in the main program. The action is the same if the variable is a parameter except that after the value is stored on the stack the local variable is initialised to the value given to the parameter by the statement that referenced the function or procedure. Once the function or procedure has finished, the original values stored in any local variable are retrieved from the stack and are returned to their original places. Any local variable without counterparts of the same name are not destroyed; they are simply left set either to zero or the null string.

If you follow the way that the BASIC stack is used every time that a function or procedure is called, you should have no trouble in following how functions and procedures can be used recursively.

Conclusion

It would be possible to write an entire book on the subject of BBC BASIC! This chapter has dealt with some of its more interesting and immediately useful aspects. Much of the information it contains can be used to write programs that not only work faster and use less memory, but are also more logical and easier to debug.