QL

Beginner's Guide

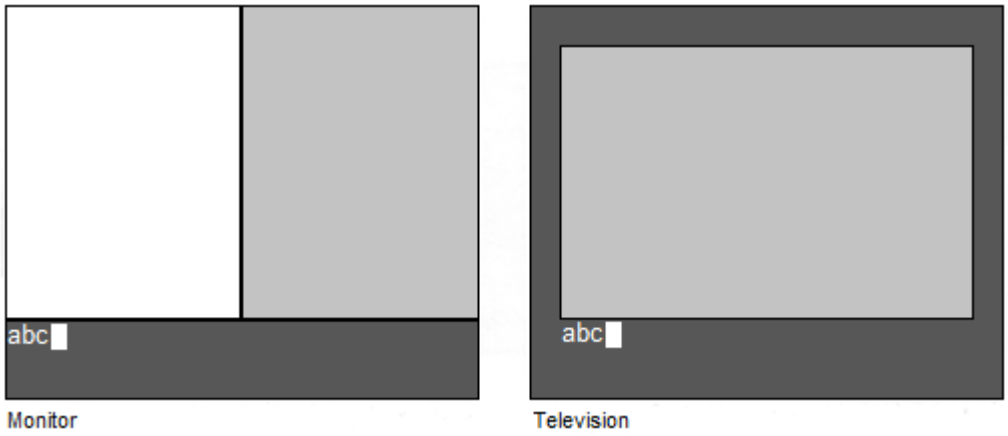
By Roy Atherton *(Bulmershe College Computer Centre)*

INDEX

CHAPTER 1 - STARTING COMPUTING

THE SCREEN

Your QL should be connected to a monitor screen or TV set and switched on. Press a few keys, say abc, and the screen should appear as shown below. The small flashing light is called the **cursor**.



If your screen does not look like this read the section entitled Introduction. This should enable you to solve any difficulties.

THE KEYBOARD

The QL is a versatile and powerful computer so there are features of the keyboard which you do not need yet. For the present we will explain just those items which you need for this and the next six chapters.

BREAK

This enables you to 'break' out of situations you do not like. For example:

a line which you have decided to abandon
something wrong which you do not understand
a running program which has ceased to be of interest
any other problem

Because BREAK is so powerful it has been made difficult to type accidentally.

Hold down **CTRL** and then press **SPACE**

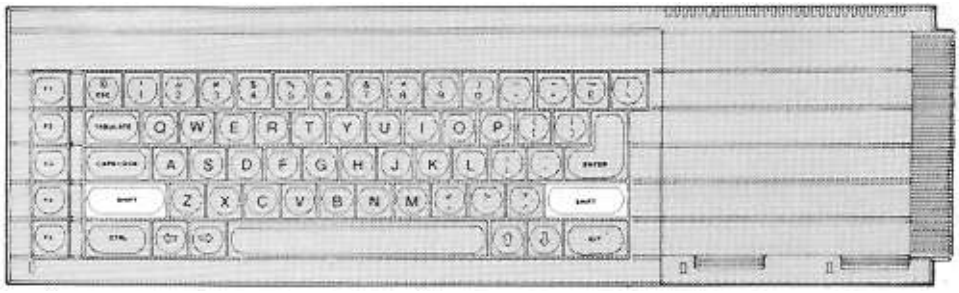
If nothing was added or removed from a program while it was halted with BREAK then it can be restarted by typing:

```
CONTINUE
```

RESET

This is not a key but a small push button on the right hand side of the QL. It is placed here deliberately, out of the way, because its effects are more dramatic than the break keys. If you cannot achieve what you need with the break keys then press the RESET Button. This is almost the same as switching the computer off and on again. You get a clean re-start.

SHIFT



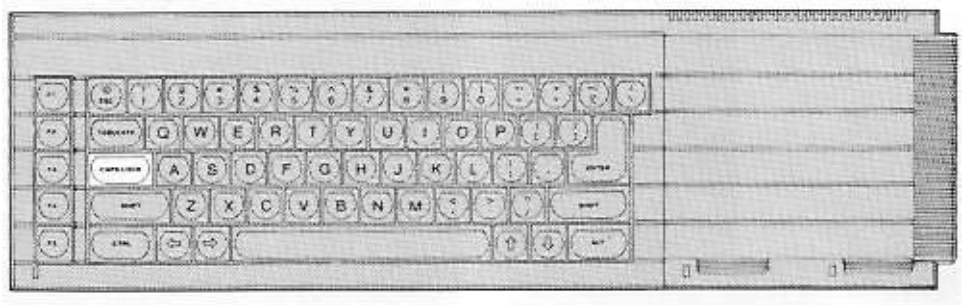
There are two **SHIFT** keys because they are used frequently and need to be available to either hand.

Hold down one **SHIFT** key and type some letter keys. You will get upper case (capital) letters.

Hold down one **SHIFT** key and type some other key not a letter. You will get a symbol in an upper position on the key.

Without a **SHIFT** key you get lower case (small) letters or a symbol in a lower position on a key.

CAPITALS LOCK

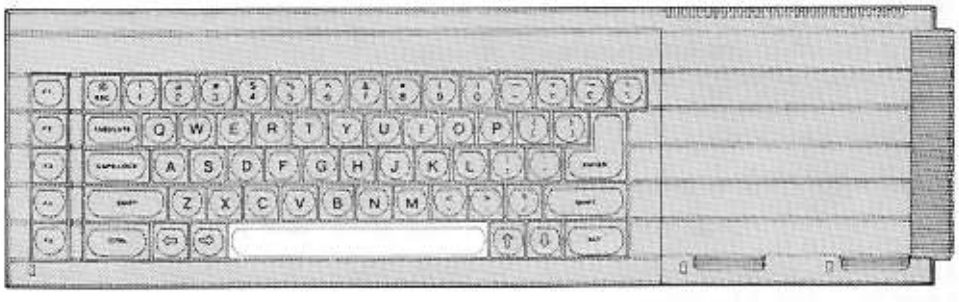


This key works like a switch Just press it once and only the letter keys will be 'locked' into a particular mode - upper case or lower case.

Type some letter keys.
Type the **CAPS LOCK** key once.
Type some letter keys.

You will see that the mode changes and remains until you type the **CAPS LOCK** key again.

SPACE BAR

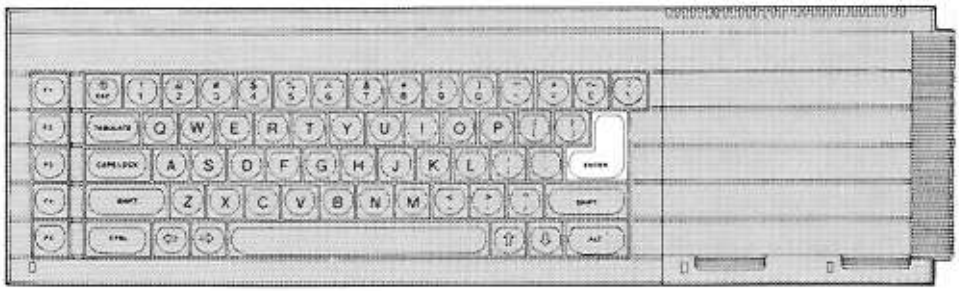


The long key at the bottom of the keyboard gives spaces. This is a very important key in SuperBASIC as you will see in chapter two.

RUBBING OUT

The left cursor together with the **CTRL** key acts like a rubber. You must hold down the **CTRL** key while you press the cursor key. Each time you then press both together the previous character is deleted.

ENTER



The system needs to know when you have typed a complete message or instruction. When you have typed something complete such as **RUN** you type the **ENTER** key to enter it into the system for action.

Because this key is needed so often we have used a special symbol for it:

↵

We shall use this for convenience, better presentation, and to save space. Test the ↵ (ENTER) key by typing

```
PRINT "Correct" ↵
```

If you made no mistakes the system will respond with

```
Correct
```

OTHER KEYBOARD SYMBOLS OF IMMEDIATE USE

*	multiply	+	add
_	underscore	=	becomes equal to (used in LET)
“	quotes	‘	apostrophe
,	comma	!	exclamation
;	semi colon	&	ampersand
:	colon	.	decimal point or full stop
\	backslash	$	dollar
(	left bracket	)	right bracket

UPPER AND LOWER CASE

```
CLS ↵
Cls ↵
clS ↵
```

These are all correct and have the same effect. Some keywords are displayed partly, in upper case to show allowed abbreviations. Where a keyword cannot be abbreviated it is displayed completely in upper case.

USE OF QUOTES

The usual use of quotes is to define a word or sentence – a string of characters. Try:

```
PRINT "This works" ↵
```

The computer will respond with:

```
This works
```

The quotes are not printed but they indicate that some text is to be printed and they define exactly what it is - everything between the opening and closing quote marks. If you wish to use the quote symbol itself in a string of characters then the apostrophe symbol can be used instead. For example:

```
PRINT 'The quote symbol is "'
```

will work and will print

```
The quote symbol is "
```

COMMON TYPING ERRORS

The zero key is with the other numeric digits at the top of the keyboard, and is slightly thinner.

The letter 'O' key is amongst the other letters. Be careful to use the right symbol.

Similarly avoid confusion between one, amongst the digits, and the letter 'I' amongst the letters between one, amongst the digits, and the letter 'I' amongst the letters.

KEEP SHIFT DOWN

When using a **SHIFT** key hold it down while you type the other key so that the **SHIFT** key makes contact before the other key and also remains in contact until after the other key has lifted.

The same rule applies to the control CTRL and alternate ALT keys which are used in conjunction with others but you do not need those at present.

Type the two simple instructions:

```
CLS ↵
PRINT 'Hello' ↵
```

Strictly speaking these constitute a computer program, however it is the stored program that is important in computing. The above instructions are executed instantly as you type ↵ (ENTER)

Now type the program with line numbers:

```
10 CLS ↵
20 PRINT 'HELLO' ↵
```

This time nothing happens externally except that the program appears in the upper part of the screen This means that it is accepted as correct grammar or syntax. It conforms to the rules of SuperBASIC, but it has not yet been executed, merely stored. To make it work, type:

```
RUN ↵
```

The distinction between direct commands for immediate action and a stored sequence of instructions is discussed in the next chapter. For the present you can experiment with the above ideas and two more:

```
LIST ↵
```

causes an internally stored program to be displayed (listed) on the screen or elsewhere.

```
NEW ↵
```

causes an internally stored program to be deleted so that you can type in a NEW one.

SELF TEST ON CHAPTER 1

You can score a maximum of 16 points from the following test. Check your score with the answers page at the end of this Beginner's Guide.

1. In what circumstances might you use the BREAK sequence?

2. Where is the RESET button?

3. What is the effect of the RESET button?

4. Name two differences between a SHIFT key and the CAPS LOCK key.

5. How can you delete a wrong character which you have just typed?

6. What is the purpose of the ENTER key?

7. What symbol do we use for the ENTER key?

What is the effect of the commands in questions 8 to 11

8. `CLS ↵`

9. `RUN ↵`

10. `LIST ↵`

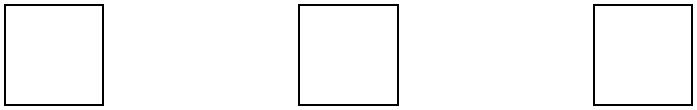
11. `NEW ↵`

12. Do keywords have the proper effect if you type them in lower case?

13. What is the significance of the parts of keywords which the QL displays in upper case?

CHAPTER 2 - INSTRUCTING THE COMPUTER

Computers need to store data such as numbers. The storage can be imagined as pigeon holes.



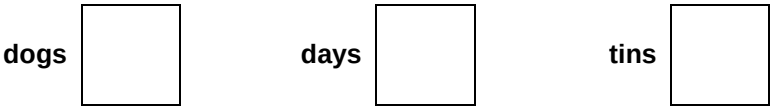
Though you cannot see them, you do need to give names to particular pigeon holes. Suppose you want to do the following simple calculation.

A dog breeder has 9 dogs to feed for 28 days, each at the rate of one tin of 'Beefo' per day. Make the computer print (display on the screen) the required number of tins.

One way of solving this problem would require three pigeon holes for

number of *dogs*
number of *days*
total number of *tins*

SuperBASiC allows you to choose sensible names for pigeon holes and you may choose as shown:



You can make the computer set up a pigeon hole name it, and store a number in it with a single instruction or statement such as:

```
LET dogs = 9 ↵
```

This will set up an internal pigeon hole named dogs, and place in it the number 9 thus:

The word **LET** has a special meaning to SuperBASIC. It is called a **keyword**. SuperBASIC has many other keywords which you will see later. You must be careful about the space after **LET** and other keywords. Because SuperBASIC allows you to choose pigeon hole names with great freedom *LETdogs* would be a valid pigeon hole name.

The **LET** keyword is optional In SuperBASIC and because of this statements like

```
LETdogs = 9 ↵
```

are valid. This would refer to a pigeon hole called **LETdogs**

Just as in English, names, numbers and keywords should be separated from each other by spaces If they are not separated by special characters.

Even if it were not necessary, a program line without proper spacing is bad style. Machines with small memory size may force programmers into it, but that is not a problem with the QL You can check that a pigeon hole exists internally by typing:

```
PRINT dogs ↵
```

The screen should display what is in the pigeon hole:

```
9
```

Again be careful to put a space after PRINT.

To solve the problem we can write a program which is a sequence of instructions or statements. You can now understand the first two:

```
LET dogs = 9 ↵
LET days = 28 ↵
```

These cause two pigeon holes to be set up, named, and given numbers or values. The next instruction must perform a multiplication, for which the computer's symbol is *, and place the result in a new pigeon hole called *tins* thus:

```
LET tins = dogs * days ↵
```

1. The computer gets the values, 9 and 28, from the two pigeon holes named dogs and days
2. The number 9 is multiplied by 28.
3. A new pigeon hole is set up and named tins.
4. The result of the multiplication becomes the value in the pigeon hole named tins.

All this may seem elaborate but you need to understand the ideas, which are very important.

The only remaining task is to make the computer print the result which can be done by typing

```
PRINT tins↵
```

which will cause the output:

```
252
```

to be displayed on the screen.

In summary the program:

```
LET dogs = 9
LET days = 28
LET tins = dogs * days
PRINT tins
```

causes the internal effects best imagined as three named pigeon holes containing numbers:

and the output on the screen:

```
252
```

Of course, you could achieve this result more easily with a calculator or a pencil and paper You could do it quickly with the QL by typing:

```
PRINT 9 * 28 ↵
```

which would give the answer on the screen. However the ideas we have discussed are the essential starting points of programming in SuperBASIC. They are so essential that they occur in many computer languages and have been given special names.

1. Names such as dogs, days and tins are called identifiers.

2. A single instruction such as:
   `LET dogs = 9` ↵
   is called a **statement**.

3. The arrangement of name and associated pigeon hole is called a **variable**. The execution of the above statement stores the value 9 in the pigeon hole 'identified' by the Identifier *dogs.*

A statement such as:

```
LET dogs = 9 ↵
```

is an instruction for a highly dynamic internal process but the printed text is static and it uses the = sign borrowed from mathematics. It is better to think or say (but not type):

```
LET dogs become 9 ↵
```

and to think of the process having a right to left direction (do not type this):

```
dogs ← 9
```

The use of = in a LET statement is not the same as the use of = in mathematics. For example, if another dog turns up you may wish to write:

```
LET dogs = dogs + 1 ↵
```

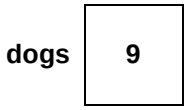
Mathematically this is not very sensible but in terms of computer operations it is simple. If the value of dogs before the operation was 9 then the value after the operation would be 10. Test this by typing:

```
LET dogs = 9 ↵
PRINT dogs ↵
LET dogs = dogs + 1 ↵
PRINT dogs ↵
```

The output should be:

```
9
10
```

proving that the final value in the pigeon hole is as shown:



A good way to understand what is happening to the pigeon holes, or variables, is to do what is called a "dry run". You simply examine each instruction in turn and write down the values which result from each instruction to show how the pigeon holes are set up and given values, and how they retain their values as the program is executed.

```
LET dogs = 9 ↵
LET days = 28 ↵
LET tins = dogs * days ↵
PRINT tins↵
```

The output should be

```
252
```

You may notice that so far a variable name has always been used first on the left hand side of a LET statement. Once the pigeon hole is set up and has a value, the corresponding variable name can be used on the right hand side of a LET statement.

Now suppose you wish to encourage a small child to save money. You might give two bars of chocolate for every pound saved. Suppose you try to compute this as follows:

```
LET bars = pounds * 2 ↵
PRINT bars ↵
```

You cannot do a dry run as the program stands because you do not know how many pounds have been saved.

We have made a deliberate error here in using pounds on the right of a LET statement without it having been set up and give n some value. Your QL will search internally for the variable "pounds". It will not find it, so it concludes that there is an error in the program and gives an error message. If we had tried to print out the value of "pounds", the QL would have printed a * to indicate that "pounds" was undefined. We say that the variable pounds has not been initialised (given an initial value). The program works properly if you do this first.

	bars	pounds
`LET pounds = 7 ↵`	7	
`LET bars = pounds * 2 ↵`	7	14

The program works properly and gives the output:

```
14
```

A STORED PROGRAM

Typing statements without line numbers may produce the desired result but there are two reasons why this method, as used so far, is not satisfactory except as a first introduction.

1. The program can only execute as fast as you can type. This is not very impressive for a machine that can do millions of operations per second.
2. The individual instructions are not stored after execution so you cannot run the program again or correct an error without re-typing the whole thing.

Charles Babbage, a nineteenth century computer pioneer knew that a successful computer needed to store instructions as well as data in internal pigeon holes. These instructions would then be executed rapidly in sequence without further human intervention.

The program instructions will be stored but not executed if you use line numbers. Try this:

```
10 LET price = 15 ↵
20 LET pens = 7 ↵
30 LET cost = price * pens ↵
40 PRINT cost ↵
```

Nothing happens externally yet, but the whole program is stored internally. You make it work by typing:

```
RUN ↵
```

and the output:

```
105
```

should appear.

The advantage of this arrangement is that you can edit or add to the program with minimal extra typing.

EDITING A PROGRAM

Later you will see the full editing features of SuperBASIC but even at this early stage you can do three things easily:

replace a line
insert a new line
delete a line

Replace a line

Suppose you wish to alter the previous program because the price has changed to 20p for a pen. Simply re-type line 10.

```
10 LET price = 20 ↵
```

This line will replace the previous line 10. Assuming the other lines are still stored, test the program by typing:

```
RUN ↵
```

and the new answer, 140, should appear.

Insert a new line

Suppose you wish to insert a line just before the last one, to print the words 'Total Cost.' This situation often arises so we usually choose line numbers 10, 20, 30 ... to allow space to insert extra lines.

To put in the extra line type

```
35 PRINT "Total Cost" ↵
```

and it will be inserted just before line 40. The system allows line numbers in the range 1 to 32768 to allow plenty of flexibility in choosing them. It is difficult to be quite sure in advance what changes may be needed.

Now type:

```
RUN ↵
```

and the new output should be:

```
Total cost
140
```

Delete Line

You can delete line 35 by typing:

```
35 ↵
```

It is as though an empty line has replaced the previous one.

OUTPUT- PRINT

Note how useful the PRINT statement is. You can PRINT text by using quotes or apostrophes:

```
PRINT "Chocolate bars" ↵
```

You can print the values of variables (contents of pigeon holes) by typing statements such as:

```
PRINT bars ↵
```

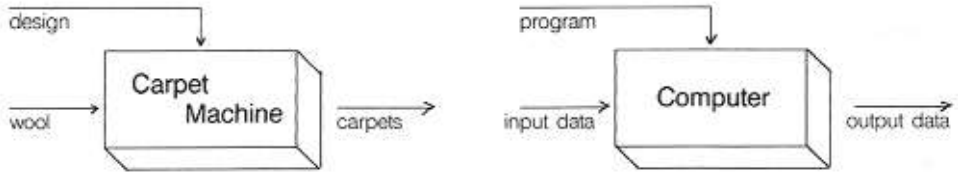
without using quotes.

You will see later how very versatile the PRINT statement can be in SuperBASIC. It will enable you to place text or other output on the screen exactly where you want it. But for the present these two facilities are useful enough:

printing of text
printing values of variables (contents of pigeon holes).

INPUT- INPUT, READ AND DATA

A carpet-making machine needs wool as input. It then makes carpets according to the current design.



If the wool is changed you may get a different carpet.

The same sort of relations exist in a computer.

However, if the data is input into pigeon holes by means of LET there are two disadvantages when you get beyond very trivial programs:

writing **LET** statements is laborious
changing such input is also laborious

You can arrange for data to be given to a program as it runs. The INPUT statement will cause the program to pause and wait for you to type in something at the keyboard. First type:

```
NEW ↵
```

so that the previous stored program (if it is still there) will be erased ready for this new one. Now type:

```
10 LET price = 15 ↵
20 PRINT "How many pens?" ↵
30 INPUT pens ↵
40 LET cost = price * pens ↵
50 PRINT cost ↵
RUN ↵
```

The program pauses at line 30 and you should type the number of pens you want, say:

```
4 ↵
```

Do not forget the ENTER key. The output will be:

```
60
```

The **INPUT** statement needs a variable name so that the system knows where to put the data which comes in from your typing at the keyboard. The effect of line 30 with your typing is the same as a **LET** statement's effect. It is more convenient for some purposes when interaction between computer and user is desirable. However, the **LET** statement and the **INPUT** statement are useful only for modest amounts of data. We need something else to handle larger amounts of data without pauses in the execution of the program.

SuperBASIC, like most BASICs, provides another method of input known as **READ**ing from **DATA** statements. We can retype the above program in a new form to give the same effects without any pauses. Try this:

```
NEW ↵
10 READ price, pens ↵
20 LET cost = price * pens ↵
30 PRINT cost ↵
40 DATA 15, 4 ↵
RUN ↵
```

The output should be:

```
60
```

as before.

Each time the program is run, SuperBASIC needs to be told where to start reading **DATA** from. This can either be done by typing **RESTORE** followed by the **DATA** line number or by typing **CLEAR**. Both these commands can also be inserted at the start of the programs.

When line 10 is executed the system searches the program for a **DATA** statement. It then uses the values in the **DATA** statement for the variables in the **READ** statement in exactly the same order. We usually place **DATA** statements at the end of a program. They are used by the program but they are not executed in the sense that every other line is executed in turn. **DATA** statements can go anywhere in a program but they are best at the end, out of the way. Think of them as necessary to, but not really part of, the active program. The rules about **READ** and **DATA** are as follows:

1. All **DATA** statements are considered to be a single long sequence of items. So far these items have been numbers but they could be words or letters.
2. Every time a **READ** statement is executed the necessary items are copied from the **DATA** statement into the variables named in the **READ** statement.
3. The system keeps track of which items have been **READ** by means of an internal record. If a program attempts to **READ** more items than exist in all the **DATA** statements an error will be signalled.

IDENTIFIERS (NAMES)

You have used names for 'pigeon holes' such as "dogs", "bars". You may choose words like these according to certain rules:

A name cannot include spaces.

A name must start with a letter.

A name must be made up from **letters, digits, $, %,** _ (underscore)

The symbols **$**, **%** have special purposes, to be explained later, but you can use the underscore to make names such as:

```
dog_food
month_wage_total
```

more readable.

SuperBASIC does not distinguish between upper and lower case letters, so names like **TINS** and **tins** are the same.

The maximum number of characters in a name is 255.

Names which are constructed according to these rules are called **identifiers**. Identifiers are used for other purposes in SuperBASIC and you need to understand them. The rules allow great freedom in choice of names so you can make your programs easier to understand. Names like *total, count, pens* are more helpful than names like Z, P, Q.

SELF TEST ON CHAPTER 2

You can score a maximum of 21 points from this test Check your score with the answers in "Answers To Self Test" section at the end of this Beginner's Guide.

1. How should you imagine an internal number store?
2. State two ways of storing a value in an internal 'pigeon hole' to be created (two points)
3. How can you find out the value of an internal 'pigeon hole'?
4. What is the usual technical name for a 'pigeon hole'?
5. When does a pigeon hole get its first value?
6. A variable is so called because its value can vary as a program is executed. What is the usual way of causing such a change?
7. The = sign in a **LET** statement does not mean 'equals' as in mathematics. What does it mean?
8. What happens when you **ENTER** an unnumbered statement?
9. What happens when you **ENTER** a numbered statement?
10. What is the purpose of quotes in a **PRINT** statement?
11. What happens when you do not use quotes in a **PRINT** statement?
12. What does an **INPUT** statement do which a **LET** statement does not?
13. What type of program statement is never executed?
14. What is the purpose of a **DATA** statement?
15. What is another word for the name of a pigeon hole (or variable)?
16. Write down three valid identifiers which use letters, letters and digits, letters and underscore (three points)
17. Why is the space bar especially important in SuperBASIC?

18. Why are freely chosen identifiers important in programming?

PROBLEMS ON CHAPTER 2

1. Carry out a dry run to show the values of all variables as each line of the following program is executed:

```
10 LET hours = 40 ↵
20 LET rate = 31 ↵
30 LET wage = hours * rate ↵
40 PRINT hours, rate, wage ↵
```

2. Write and test a program, similar to that of problem 1, which compute s the area of a carpet is 3 metres in width and 4 metres in length. Use the variable names: *width, length, area*.

3. Re-write the program of problem 1 so that it uses two **INPUT** statements instead of **LET** statements.

4. Re write the program of problem 1 so that the input data (40 and 3) appears in a DATA statement instead of a **LET** statement.

5. Re write the program of problem 2 using a different method of data input. Use **READ** and **DATA** if you originally used **LET** and vice-versa.

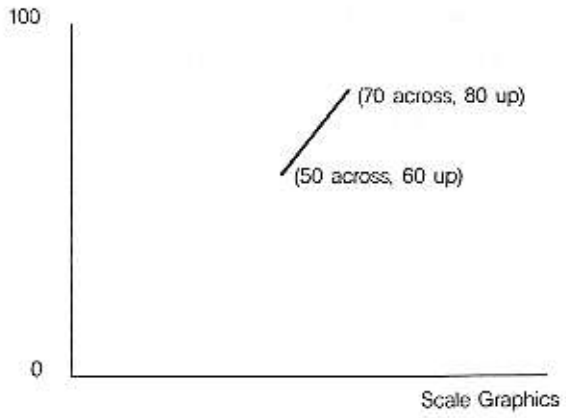
6. Bill and Ben agree to have a gamble. Each will take out of his wallet all the pound notes and give them to the other. Write a program to simulate this entirely with **LET** and **PRINT** statements. Use a third person, Sue, to hold Bill's money while he accepts Ben's.

7. Re-write the program of problem 6 so that a DATA statement holds the two numbers to be exchanged.

CHAPTER 3 - DRAWING ON THE SCREEN

In order to use either a television set or monitor with the QL, two different screen modes are available. MODE 8 permits eight colour displays with a graphics resolution of 256 by 256 pixels and large characters for display on a television set. MODE 4 allows four colours with a resolution of 512 by 256 pixels and a maximum of eighty character lines for which a monitor must be used for successful display. However, it would be unfortunate if a program was written to draw circles or squares in one mode and produced ellipses or rectangles in another mode (as some systems do). We therefore provide a system of scale graphics which avoids these problems. You simply choose a vertical scale and work to it. The other type of graphics (pixel oriented) is also available and is described fully in a later chapter.

Suppose, for example, that we choose a vertical scale of 100 and we wish to draw a line from position (50,60) to position (70,80).



A COLOURED LINE

We need to specify three things:

PAPER (background colour)
INK (drawing colour)
LINE (start and end points)

The followingprogram will draw a line as shown in the above figure in red (colour code 2) on a white (colour code 7) background.

```
NEW ↵
10 PAPER 7 : CLS ↵
20 INK 2 ↵
30 LINE 50,60 TO 70,80 ↵
RUN ↵
```

In line 10 the paper colour is selected first but it only comes into effect with a further command, such as **CLS**, meaning clear the screen to the current paper colour.

MODES AND COLOURS

So far it does not matter which screen mode you are using but the range of colours is affected by the choice of mode.

MODE 8 allows eight basic colours
MODE 4 allows four basic colours

Colours have codes as described below.

Code	Effect	
	8 colour	4 colour
0	black	black
1	blue	black
2	red	red
3	magenta	red
4	green	green
5	cyan	green
6	yellow	white
7	white	white

For example, **INK 3** would give magenta in **MODE 8** and red in **MODE 4**.

We will explain in a later chapter how the basic colours can be 'mixed' in various ways to produce a startling range of colours, shades and textures.

RANDOM EFFECTS

You can get some interesting effects with random numbers which can be generated with the RND function. For example:

```
PRINT RND (1 TO 6)  ↵
```

will print a whole number in the range 1 to 6, like throwing an ordinary six-sided dice. The following program will illustrate this:

```
NEW ↵
10 LET die = RND(1 TO 6) ↵
20 PRINT die ↵
RUN ↵
```

If you run the program several times you will get different numbers.

You can get random whole numbers in any range you like. For example:

```
RND(0 TO 100)
```

will produce a number which can be used in scale graphics. You can re-write the line program so that it produces a random colour. Where the range of random numbers starts from zero you can omit the first number and write:

```
RND(100)

NEW ↵
10 PAPER 7 : CLS ↵
20 INK RND(5) ↵
30 LINE 50,60 TO RND(100),RND(100) ↵
RUN ↵
```

This produces a line starting somewhere near the centre of the screen and finishing at some random point. The range of possible colours depends on which mode is selected. You will find that a range of numbers ‘something TO something’ occurs often in SuperBASIC.

BORDERS

The part of the screen in which you have drawn lines and create other output is called a window. Later you will see how you can change the size and position of a window or create other windows. For the present we shall be content to draw a border round the current window. The smallest area of light or colour you can plot on the screen is called a pixel. In mode 8, called low resolution mode, there are 256 possible pixel positions across the screen and 256 down. In mode 4, called high resolution mode, there are 512 pixels across the screen and 256 down. Thus the size of a pixel depends on the mode.

You can make a border round the inside edge of a window by typing for example:

```
BORDER 4,2 ↵
```

This will create a border 4 pixels wide in colour red (code 2). The effective size of the window is reduced by the border. This means that any subsequent printing or graphics will automatically fit within the new window size. The only exception to this is a further border which will overwrite the existing one.

A SIMPLE LOOP

Computers can do things very quickly but it would not be possible to exploit this great power if every action had to be written as an instruction. A building foreman has a similar problem. If he wants a workman to lay a hundred paving stones that is roughly what he says. He does not give a hundred separate instructions.

A traditional way of achieving looping or repetition in BASIC is to use a **GO TO** (or **GOTO**, they are the same) statement as follows.

```
NEW ↵
10 PAPER 6 : CLS ↵
20 BORDER 1,2 ↵
30 INK RND(5) ↵
40 LINE 50,60 TO RND(100),RND(100) ↵
50 GOTO 30 ↵
RUN ↵
```

You may prefer not to type in this program because SuperBASIC allows a better way of doing repetition. Note certain things about each line.

10 20	Fixed part – not repeatd
30 40	Changeable part – repeated
50	Controls program

You can re-write the above program by omitting the **GOTO** statement and, instead, putting **REPeat** and **END REPeat** around the part to be repeated.

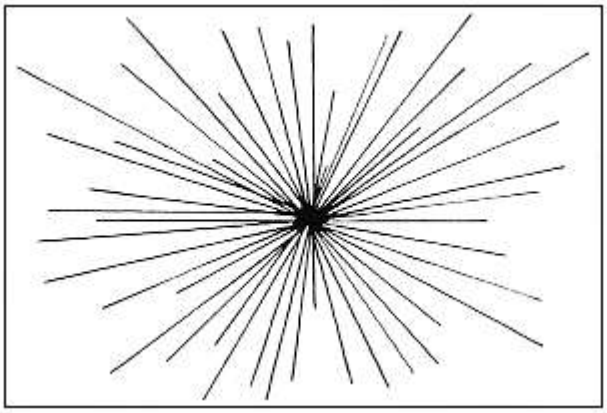
```
NEW ↵
10 PAPER 6 : CLS ↵
20 BORDER 1,2 ↵
30 REPEAT star ↵
40   INK RND(5) ↵
50   LINE 50,60 TO RND(100),RND(100) ↵
60 END REPEAT star ↵
RUN ↵
```

We have given the *repeat structure* a name, *star*. The structure consists of the two lines:

```
REPeat star
END REPeat star
```

and what lies between them is called the content of the structure. The use of upper case letters indicates that REP is a valid abbreviation of REPeat.

This program should produce coloured lines indefinitely to make a star as shown in the figure below.



The STAR program

You can stop it by pressing the break keys:

Hold down CTRL and then press SPACE.

SuperBASIC provides a consistent and versatile method of stopping repetitive processes. Imagine running round and round inside the program activating statements. How can you escape? The answer is to use an **EXIT** statement. But there must be some reason for escaping. You might extend the choice of line colours by typing as an amendment to the program (do not type **NEW**):

```
40 INK RND (0 TO 6) ↵
```

so that if **RND** produces 6 the ink is the same colour as the paper and you will not see it. This could be the reason for terminating the repetition. We can re-arrange the program as follows:

```
NEW ↵
10 PAPER 6 : CLS ↵
20 BORDER 1 ,2 ↵
30 REPeat star ↵
40   LET colour = RND(6) ↵
50   IF colour = 6 THEN EXIT star ↵
60   INK colour ↵
70   LINE 50,60 TO RND(100),RND(100) ↵
80 END REPeat star ↵
```

The important thing to note here is that the program continues until "colour" becomes 6. Control then escapes from the loop to the point just after line 80. Since there are no program lines after 80 the program stops.

Another important concept has been introduced. It is the idea of a decision.

```
IF colour = 6 THEN EXIT star
```

This is another very useful structure because it is a choice of doing something or not; we call it a simple binary decision. Its general form is:

```
IF condition THEN statement(s)
```

You will see later how the two concepts of repetition (or looping) and decision-making (or selection) are the main structures for program control. You can stop the program by pressing the break keys: hold down **CTRL** and then press the space bar.

SELF TEST ON CHAPTER 3

You can score a maximum of 13 points from the following test. Check your score with the answers on Page 107 - in the "Answers to self test" section at the end of this Beginner's Guide.

1. What is a pixel?

2. How many pixels fit across the screen in the low resolution mode?

3. How many pixels fit from bottom to top in low resolution mode?

4. What are the two numbers which determine the 'address' or position of a graphics point on the screen?

5. How many colours are available in the low resolution mode?

6. Name the keywords which do the following:
   i. draw a line
   ii. select a colour for drawing
   iii. iii select a background colour
   iv. draw a border (5 points)

7. What are the statements which open and close the REPeat loop?

8. When does an executing REPeat loop terminate?

9. Why do loops in SuperBASIC have names?

PROBLEMS ON CHAPTER 3

1. Write a program to draw straight lines all over the screen. The lines should be of random length and direction. Each should start where the previous one finished and each should have a randomly chosen colour.

2. Write a program to draw lines randomly with the restriction that each line has a random start on the left hand edge of the screen.

3. Write a program to draw lines randomly with the restriction that the lines start at the same point on the bottom edge of the screen.

4. Write a program to produce lines of random length, starting points and colour. All lines must be horizontal.

5. As problem 4 but make the lines vertical.

6. Write a program to produce a square 'spiral' in such a way that each line makes a random colour

HINT: First find the co-ordinates of some of the corners, then put them in groups of four. You should discover a pattern.

CHAPTER 4 – CHARACTERS AND STRINGS

Teachers sometimes wish to assess the reading ability needed for particular books or classroom materials. Various tests are used and some of these compute the average lengths of words and sentences. We will introduce ideas about handling words or **character strings** by examining simple approaches to finding average word lengths.

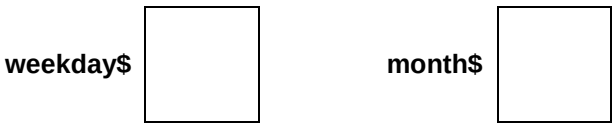
We are talking about sequences of letters, digits or other symbols which may or may not be words. That is why the term 'character string' has been invented. It is usually abbreviated to **string**. Strings are handled in ways similar to number handling but, of course, we do not do the same operations on them. We do not multiply or subtract strings. We join them, separate them, search them and generally manipulate them as we need.

NAMES AND PIGEON HOLES FOR STRINGS

You can create pigeon holes for strings. You can put character strings into pigeon holes and use the information just as you do with numbers. If you intend to store (not all at once) words such as:

FIRST SECOND THIRD
and
JANUARY FEBRUARY MARCH

you may choose to name two pigeon holes:



Notice the dollar sign. Pigeon holes for strings are internally different from those for numbers and SuperBASIC needs to know which is which. All names of string pigeon holes must end with **$**. Otherwise the rules for choosing names are the same as the rules for the names of numeric pigeon holes.

You may pronounce:

weekday$ as weekdaydollar
month$ as monthdollar

The LET statement works in the same way as for numbers. If you type:

```
LET weekday$ = "FIRST" ↵
```

an internal pigeon hole, named *weekday$* will be set up with the value FIRST in it thus:

The quote marks are not stored. They are used in the LET statement to make it absolutely clear what is to be stored in the pigeon hole. You can check by typing:

```
PRINT weekday$ ↵
```

and the screen should display what is in the pigeon hole:

```
FIRST
```

You can use a pair of apostrophes instead of a pair of quote marks.

LENGTHS OF STRINGS

SuperBASIC makes it easy to find the length or number of characters of any string. You simply write, for example:

```
PRINT LEN(weekday$) ↵
```

If the pigeon hole, weekday$, contains FIRST the number 5 will be displayed. You can see the effect in a simple program:

```
NEW ↵
10 LET weekday$ = "FIRST" ↵
20 PRINT LEN(weekday$) ↵
RUN ↵
```

The screen should display:

```
5
```

LEN is a keyword of SuperBASIC

An alternative method of achieving the same result uses both a string pigeon hole and a numeric pigeon hole.

```
NEW ↵
10 LET weekday$ = "FIRST" ↵
20 LET length = LEN(weekday$) ↵
30 PRINT length ↵
RUN ↵
```

The screen should display:

```
5
```

as before, and two internal pigeon holes contain the values shown:

Let us return to the problem of average lengths of words.

Write a program to find the average length of the three words:

```
FIRST, OF, FEBRUARY
```

PROGRAM DESIGN

When problems get beyond what you regard as very trivial, it is a good idea to construct a **program design** before writing the program itself

1. Store the three words in pigeon holes.
2. Compute the lengths and store them.
3. Compute the average.
4. Print the result.

```
NEW ↵
```

```
10 LET weekday$ = "FIRST" ↵
20 LET word$ = "OF" ↵
30 LET month$ = "FEBRUARY" ↵
40 LET length1 = LEN (weekday$) ↵
50 LET length2 = LEN (word$) ↵
60 LET length3 = LEN (month$) ↵
70 LET sum = length1 + length2 + length3 ↵
80 LET average = sum/3 ↵
90 PRINT average ↵
RUN ↵
```

The symbol **/** means **divided by**. The output or result of running the program is simply:

```
5
```

And there are eight internal pigeon holes involved:

weekday$	FIRST	length1	5
word$	OF	length2	2
month$	FEBRUARY	length3	3
		sum	15
		average	5

If you think that is a lot of fuss for a fairly simple problem you can certainly shorten it. The shortest version would be a single line but it would be less easy to read. A reasonable compromise uses the symbol "&" which stands for the operation:

```
Join two strings
```

Now type:

```
NEW ↵
10 LET weekday$ = "FIRST" ↵
20 LET word$ = "OF" ↵
30 LET month$ = "FEBRUARY" ↵
40 LET phrase$ = weekday$ & word$ & month$ ↵
50 LET length = LEN(phrase$) ↵
60 PRINT length/3 ↵
RUN ↵
```

The output is 5 as before but there are some different internal effects:

Weekday$	FIRST	Length	15

Word$	OF
Month	FEBRUARY
Phrase$	FIRSTOFFEBRUARY

There is one more reasonable simplification which is to use READ and DATA instead of the first three LET statements. Type:

```
NEW ↵
10 READ weekday$, word$, month$ ↵
20 LET phrase$ = weekday$ & word$ & month$ ↵
30 LET length = LEN(phrase$) ↵
40 PRINT length/3 ↵
50 DATA "FIRST","OF","FEBRUARY" ↵
RUN ↵
```

The internal effects of this version are exactly the same as those of the previous one. **READ** causes the setting up of internal pigeon holes with values in them in a similar way to **LET**.

IDENTIFIERS AND STRING VARIABLES

Names of pigeon holes, such as:

weekday$
word$
month$
phrase$

are called **string identifiers**. The dollar signs imply that the pigeon holes are for character strings. The dollar must always be at the end.

Pigeon holes of this kind are called **string variables** because they contain only character strings which may vary as a program runs.

The contents of such pigeon holes are called values. Thus words like 'FIRST' and 'OF' may be values of string variables named *weekday$* and *+word$*

RANDOM CHARACTERS

You can use character codes (see Concept Reference Guide) to generate random letters. The upper case letters A to Z have the codes 65 to 90. The function CHR$ converts these codes into letters. The following program will print a letter B.

```
NEW ↵
10 LET lettercode = 66 ↵
20 PRINT CHR$ (lettercode) ↵
RUN ↵
```

The following program will generate trios of letters A, B, or C until the word CAB is spelled accidentally.

```
NEW ↵
10 REPeat taxi
20 LET first$ = CHR$(RND(65 TO 67))
30 LET second$ = CHR$(RND(65 TO 67))
```

```
40 LET third$ = CHR$(RND(65 TO 67))
50 LET word$ = first$ & second$ & third$
60 PRINT ! word$ !
70 IF word$ = "CAB" THEN EXIT taxi
80 END REPeat taxi
```

Random characters, like random numbers or random points are useful for learning to program. You can easily get interesting effects for program examples and exercises.

Note the effect the ! … ! have on the spacing of the output.

SELF TEST ON CHAPTER 4

You can score a maximum of 10 points from the following test. Check your score with the answers in the "Answers To Self Tests" section at the end of this Beginner's Guide.

1. What is a character string?
2. What is the usual abbreviation of the term, 'character string'?
3. What distinguishes the name of a string variable?
4. How do some people pronounce a word such as 'word$'?
5. What keyword is used to find the number of characters in a string?
6. What symbol is used to join two strings?
7. Spaces can be part of a string. How are the limits of a string defined?
8. When a statement such as:
   ```
   LET meat$ = "steak"
   ```
   is executed, are the quotes stored?
9. What function will turn a suitable code number into a letter?
10. How can you generate random upper case letters?

PROBLEMS ON CHAPTER 4

1. Store the words 'Good' and 'day' in two separate variables. Use a LET statement to join the values of the two variables in a third variable. Print the result.
2. Store the following words in four separate pigeon holes:

   ```
   light  let  be  there
   ```

   Join the words to make a sentence adding spaces and a full stop. Store the whole sentence in a variable, *sent$*, and print the sentence and the total number of characters it contains.
3. Write a program which uses the keywords:

   ```
   CHR$ RND(65 TO 90))
   ```

   to generate one hundred random three letter words. See if you have accidentally generated any real English words. Test the effects of:

a) ; at the end of a PRINT statement.
b) ! on either side of item printed.

CHAPTER 5 - KNOWN GOOD PRACTICE

You have already begun to work effectively with short programs. You may have found the following practices are helpful:

1. Use of lower case for identifiers: names of variables (pigeon holes) or repeat structures, etc.
2. Indenting of statements to show the content of a repeat structure.
3. Well chosen identifiers reflecting what a variable or repeat structure is used for.
4. Editing a program by:

    replacing a line
    inserting a line
    deleting a line

PROGRAMS AS EXAMPLES

You have reached the stage where it is helpful to be able to study programs to learn from them and to try to understand what they do. The mechanics of actually running them should now be well understood and in the following chapters we will dispense with the constant repetition of:

```
NEW before each program
↵ at the end of each line
RUN to start each program
```

You will understand that you should use all these features when you wish to enter and run a program. But their omission in the text will enable you to see the other details more clearly as you try to imagine what the program will do when it runs.

If we dispense with the above details we may use and understand programs more easily without the technical clutter. For example, the following program generates random upper case letters until a Z appears. It does not show the words **NEW** or **RUN** or the **ENTER** symbol but you still need to use these.

```
10 REPeat letters
20 LET lettercode = RND(65 TO 90)
30 cap$ = CHR$(lettercode)
40 PRINT cap$
50 IF cap$ = "Z" THEN EXIT letters
60 END REPeat letters
```

In this and subsequent chapters programs will be shown without **ENTER** symbols. Direct commands will also be shown without **ENTER** symbols. But you must use these keys as usual. You must also remember to use **NEW** and **RUN** as necessary

AUTOMATIC LINE NUMBERING

It is tedious to enter line numbers manually. Instead you can type:

```
AUTO
```

before you start programming and the QL will reply with a line number:

```
100
```

Continue typing lines until you have finished your program when the screen will show:

```
100 PRINT "First"
110 PRINT "Second"
120 PRINT "End"
```

To finish the automatic production of line numbers use the BREAK sequence:

Hold down the **CTRL** and press the **SPACE** bar. This will produce the message:
`130 not complete`

and line 130 will not be included in your program.

If you make a mistake which does not cause a break from automatic numbering, you can continue and **EDIT** the line later. If you want to start at some particular line number say 600, and use an increment other than 10 you can type, for an increment of 5:

```
AUTO 600,5
```

Lines will then be numbered 600, 605, 610, etc.

To cancel **AUTO**, press **CTRL** and the **SPACE** bar at the same time.

EDITING A LINE

To edit a line simply type **EDIT** followed by the line number for example:

```
EDIT 110
```

The line will then be displayed with the cursor at the end thus:

```
110 PRINT "Second"
```

You can move the cursor using:

← one place left
→ one place right

To delete a character to the left use:

CTRL with ←

To delete the character in the cursor position type:

CTRL with →

and the character to the right of the cursor will move up to close the gap.

USING MICRODRIVE CARTRIDGES

Before using a new Microdrive cartridge it must be formatted. Follow the instructions in the *Introduction*. The choice of name for the cartridge follows the same rules as SuperBASIC identifiers, etc. but limited to only 10 characters. It is a good idea to write the name of the cartridge on the cartridge itself using one of the supplied sticky labels. You should always keep at least one back-up copy of any program or data. Follow the instructions in the *Information* section of the User Guide.

WARNING

If you FORMAT a cartridge which holds programs and/or data,
ALL the programs and/or data will be lost

SAVING PROGRAMS

The following program sets borders, 8 pixels wide, in red (code 2), in three windows designated #0, #1, #2.

```
100 REMark Border
110 FOR k = 0 TO 2 : BORDER #k,8,2
```

You can save it on a microdrive by inserting a cartridge and typing:

```
SAVE mdv1_bord
```

The program will be saved in a Microdrive file called "bord".

CHECKING A CARTRIDGE

If you want to know what programs or data files are on a particular cartridge place it in Microdrive 1 and type:

```
DIR mdv1_
```

The directory will be displayed on the screen. If the cartridge is in Microdrive 2 then type instead:

```
DIR mdv2_
```

COPYING PROGRAMS AND FILES

Once a program is stored as a file on a Microdrive cartridge it can be copied to other files. This is one way of making a backup copy of a Microdrive cartridge. You might copy all the previous programs, and similar commands for other programs, onto another cartridge in Microdrive 2 by typing:

```
COPY mdv1_bord TO mdv2_bord
```

DELETING A CARTRIDGE FILE

A file is anything, such as a program or data, stored on a cartridge. To delete a program called "prog" you type:

```
DELETE mdv1_prog
```

LOADING PROGRAMS

A program can be loaded from a Microdrive cartridge by typing:

```
LOAD mdv2_bord
```

If the program loads correctly it will prove that both copies are good. You can test the program by using:

LIST to list it.
RUN to run it.

Instead of using LOAD followed by RUN you can combine the two operations in one command.

```
LRUN mdv2_bord
```

The program will load and execute immediately.

MERGING PROGRAMS

Suppose that you have two programs saved on Microdrive 1 as "prog1" and "prog2".

```
100 PRINT "First"
110 PRINT "Second"
```

If you type:

```
LOAD mdv1_prog1
```

followed by:

```
MERGE mdv1_prog2
```

The two programs will be merged into one. To verify this, type LIST and you should see:

```
100 PRINT "First"
110 PRINT "Second"
```

If you MERGE a program make sure that all its line numbers are different from the program already in main memory. Otherwise it will overwrite some of the lines of the first program. This facility becomes very valuable as you become proficient in handling procedures. It is then quite natural to build a program up by adding procedures or functions to it.

GENERAL

Be careful and methodical with cartridges. Always keep one back-up copy and if you suspect any problem with a cartridge or microdrive keep a second back-up copy. Computer professionals very rarely lose data. They know that even with the best machines or devices there will be occasional faults and they allow for this.

If you want to call a program by a particular name, say, square, it may be a good idea to use names like sq1, sq2... for preliminary versions. When the program is in its final form take at least two copies called square and the others may be deleted by re-formatting or by some more selective method.

SELF TEST ON CHAPTER 5

You can score a maximum of 14 points from the following test. . Check your score with the answers in the "Answers To Self Tests" section at the end of this Beginner's Guide.

1. Why are lower case letters preferred for program words which you choose?
2. What is the purpose of indenting?
3. What should normally guide your choice of identifiers for variables and loops?
4. Name three ways of editing a program in the computer's main memory (three points).
5. What should you remember to type at the end of every command or program line when you enter it?
6. What should you normally type before you enter a program at the keyboard?
7. What must be at the beginning of every line to be stored as part of a program?
8. What must you remember to type to make a program execute?
9. What keyword enables you to put into a program information which has no effect on the execution?
10. Which two keywords help you to store programs on and retrieve from cartridges? (two points).

PROBLEMS ON CHAPTER 5

1. Re-write the following program using lower case letters to give a better presentation. Add the words **NEW** and **RUN**. Use line numbers and the **ENTER** symbol just as you would to enter and run a program. Use **REMark** to give the program a name.

```
LET TWO$ = "TWO"
LET FOUR$ = "FOUR"
LET SIX$ = TWO$ & FOUR$
PRINT LEN(six$)
```

   Explain how two and four can produce 7.

2. Use indenting, lower case letters, NEW, RUN, line numbers and the ENTER symbol to show how you would actually enter and run the following program:

```
REPEAT LOOP
LETTER_CODE = RND(65 TO 90)
LET LETTERS$ = CHR$(LETTER_CODE)
PRINT LETTER$
IF LETTER$ = 'Z' THEN EXIT LOOP
END REPEAT LOOP
```

3. Re-write the following program in better style using meaningful variable names and good presentation. Write the program as you would enter it:

```
LET S = O
REPeat TOTAL
LET N = RND(1 TO 6)
PRINT ! N !
LET S = S + N
IF n = 6 THEN EXIT TOTAL
END REPeat TOTAL
PRINT S
```

   Decide what the program does and then enter and run it to check your decision.

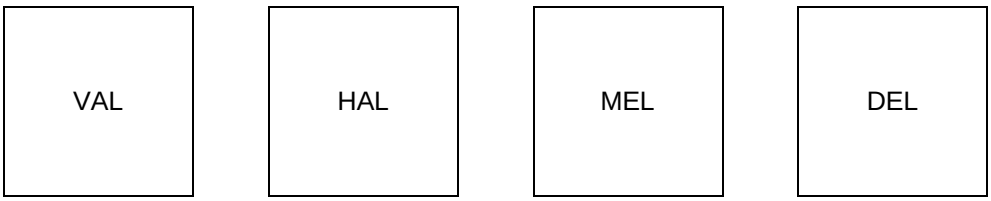
CHAPTER 6 – ARRAYS AND FOR LOOPS

WHAT IS AN ARRAY

You know that numbers or character strings can become values of variables. You can picture this as numbers or words going into internal pigeon holes or houses. Suppose for example that four employees of a company are to be sent to a small village, perhaps because oil has been discovered. The village is one of the few places where the houses only have names and there are four available for rent. All the house names end with a dollar symbol.

Westlea$ Lakeside$ Roselawn$ Oaktree$

The four employees are called:



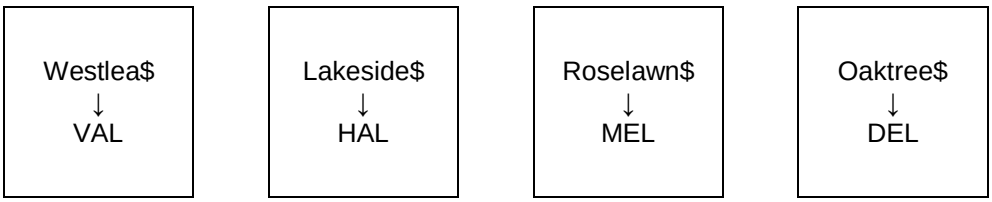
They can be placed in the houses by one of two methods.

Program 1:

```
100 LET westlea$ = "VAL"
110 LET lakeside$ = "HAL"
120 LET roselawn$ = "MEL"
130 LET oaktree$ = "DEL"
140 PRINT ! westlea$ ! lakeside$ ! roselawn$ ! oaktree$
```

Program 2:

```
100 READ westlea$, lakeside$, roselawn$, oaktree$
110 PRINT ! westlea$ ! lakeside$ ! roselawn$ ! oaktree$
120 DATA "VAL", "HAL", "MEL", "DEL"
```



As the amount of data gets larger the advantages of **READ** and **DATA** over **LET** become greater. But when the data gets really numerous the problem of finding names for houses gets as difficult as finding vacant houses in a small village.

The solution to this and many other problems of handling data lies in a new type of pigeon hole or variable in which many may share a single name. However, they must be distinct so each variable also has a number like numbered houses in the same street. Suppose that you need four vacant houses in High Street numbered 1 to 4. In SuperBASIC we say there is an **array** of four houses. The name of the array is *high_st$* and the four houses are to be numbered 1 to 4.

But you cannot just use these array variables as you can ordinary (simple)variables. You have to declare the dimensions (or size) of the array first. The computer allocates space internally and it needs to know how many string variables there are in the array and also the maximum length of each string variable. You use a **DIM** statement thus:

```
DIM high_st$(4, 3)
             |  |
             |   ------ maximum length of string
             |
              --------- number of string variables
```

After the DIM statement has been executed the variables are available for use. It is as though the houses have been built but are still empty. The four 'houses' share a common name, *high_st$*, but each has its own number and each can hold up to three characters.

There are five programs below which all do the same thing: they cause the four 'houses' to be 'occupied' and they **PRINT** to show that the 'occupation' has really worked. The final method uses only four lines but the other four lead up to it in a way which moves all the time from known ideas to new ones or new uses of old ones. The movement is also towards greater economy.

If you understand the first two or three methods perfectly well you may prefer to move straight onto methods 4 and 5. But if you are in any doubt, methods 1, 2 and 3 will help to clarify things.

Program 1

```
100 DIM high_st$(4,3)
110 LET high_st$(1) = "VAL"
120 LET high_st$(2) = "HAL"
130 LET high_st$(3) = "MEL"
140 LET high st$(4) = "DEL"
150 PRINT ! high_st$(1) ! high st$(2) !
160 PRINT ! high_st$(3) ! high-st$(4) !
```

Program 2

```
100 DIM high st$(4,3)
110 READ high_st$(1),high_st$(2),high_st$(3),high_st$(4)
120 PRINT ! high_st$(1) ! high_st$(2) !
130 PRINT ! high_st$(3) ! high_st(4) !
140 DATA "VAL","HAL","MEL","DEL"
```

This shows how to economise on variable names but the constant repeating of *high_st$* is both tedious and the cause of the cluttered appearance of the programs. We can, again, use a known technique - the **REPeat** loop to improve things further. We set up a counter, number, which increases by one as the **REPeat** loop proceeds.

Program 3

```
100 RESTORE 190
110 DIM high_st$(4,3)
120 LET number = 0
130 REPeat houses
140   LET number = number + 1
150   READ high_st$(number)
160   IF num = 4 THEN EXIT houses
170 END REPeat houses
180 PRINT high_st$(1) ! high_st$(2) ! high_st$(3) ! high_st$(4)
190 DATA "VAL","HAL","MEL","DEL"
```

:

This special type of loop, in which something has to be done a certain number of times, is well known. A special structure, called a **FOR** loop, has been invented for it. In such a loop the count from 1 to 4 is handled automatically. So is the exit when all four items have been handled.

Program 4

```
100 RESTORE 160
110 DIM high_st$(4,3)
120 FOR number = 1 TO 4
130 READ high_st$(number)
140 PRINT ! high_st$(number) !
150 END FOR number
160 DATA "VAL","HAL","MEL","DEL"
```

The output from all four programs is the same:

```
VAL HAL MEL DEL
```

Which proves that the data is properly stored internally in the four array variables:

Method 4 is clearly the best so far because it can deal equally well with 4 or 40 or 400 items by just changing the number 4 and adding more **DATA** items. You can use as many **DATA** statements as you need.

In its simplest form the **FOR** loop is rather like the simplest form of **REPeat** loop. The two can be compared:

```
100 REPeat greeting       100 FOR greeting = 1 TO 40
110 PRINT 'Hello"         110 PRINT 'Hello"
120 END REPeat greeting   120 END FOR greeting
```

Both these loops would work. The **REPeat** loop would print 'Hello' endlessly (stop it with the **BREAK** sequence) and the **FOR** loop would print 'Hello' just forty times.

Notice that the name of the **FOR** loop is also a variable, *greeting*, whose value varies from 1 to 40 in the course of running the program. This variable is sometimes called the **loop variable** or the **control variable** of the loop.

Note the structure of both loops takes the form:

```
Opening statement
  Content
Closing statement
```

However certain structures have allowable short forms for use when there are only one or a few statements in the content of the loop. Short forms of the **FOR** loop are allowed so we could write the program in the most economical form of all:

Program 5:

```
100 RESTORE 140 : CLS
110 DIM high st$(4,3)
120 FOR number = 1 TO 4 : READ high_st$(number)
```

```
130 FOR number = 1 TO 4 : PRINT ! high_st$(number) !
140 DATA "VAL", "HAL", "MEL", "DEL"
```

Colons serve as end of statement symbols instead of **ENTER** and the **ENTER** symbols of lines 120 and 130 serve as **END FOR** statements.

There is an even shorter way of writing the above program. To print out the contents of the array **high_st$** we can replace line 130 by:

```
130 PRINT ! high_st$ !
```

This uses an array slicer which we will discuss later in chapter 13.

We have introduced the concept of an array of string variables so that the only numbers involved would be the subscripts in each variable name. Arrays may be string or numeric and the following examples illustrate the numeric array.

Program 1:

Simulate the throwing of a pair of dice four hundred times. Keep a record of the number of occurrences of each possible score from 2 to 12.

```
100 REMark DICE1
110 LET two = 0  :three = 0:four = 0:five = 0:six = 0
120 LET seven = 0:eight = 0:nine = 0:ten = 0 :eleven = 0:twelve = 0
130 FOR throw = 1 TO 400
140   LET die1 = RND(1 TO 6)
150   LET die2 = RND(1 TO 6)
160   LET score = die1 + die2
170   IF score = 2 THEN LET two = two + 1
180   IF score = 3 THEN LET three = three + 1
190   IF score = 4 THEN LET four = four + 1
200   IF score = 5 THEN LET five = five + 1
21O   IF score = 6 THEN LET six = six + 1
220   IF score = 7 THEN LET seven = seven + 1
230   IF score = 8 THEN LET eight = eight + 1
240   IF score = 9 THEN LET nine = nine + 1
250   IF score = 10 THEN LET ten = ten + 1
26O   IF score = 11 THEN LET eleven = eleven + 1
270   IF score = 12 THEN LET twelve = twelve + 1
280 END FOR throw
290 PRINT ! two ! three ! four ! five ! six
300 PRINT ! seven ! eight ! nine ! ten ! eleven ! twelve
```

In the above program we establish eleven simple variables to store the tally of the scores. If you plot the tallies printed at the end you find that the bar chart is roughly triangular. The higher tallies are for scores six, seven, eight and the lower tallies are for 2 and 12. As every dice player knows, the reflects the frequency of the middle range of scores (six,seven,eight) and the rarity of twos or twelves.

```
100 REMark Dice2
110 DIM tally(12)
120 FOR throw = 1 TO 400
130 LET die_1 = RND(1 TO 6)
140 LET die_2 = RND(1 TO 6)
150 LET score = die_1 + die_2
160 LET tally(score) = tally(score) + 1
170 END FOR throw
180 FOR number = 2 TO 12 : PRINT tally(number)
```

In the first FOR loop, using *throw*, the subscript of the array variable is *score*. This means that the correct array subscript is automatically chosen for an increase in the tally after each throw. You can

think of the array, **tally**, as a set of pigeon-holes numbered 2 to 12. Each time a particular score occurs the tally of that score is increased by throwing a stone into the corresponding pigeon hole.

In the second (short form) **FOR** loop, the subscript is *number*. As the value of *number* changes from 2 to 12 all the values of the tallies are printed.

Notice that in the **DIM** statement for a numeric array you need only declare the number of variables required. There is no question of maximum length as there is in a string array.

If you have used other versions of BASIC you may wonder what has happened to the **NEXT** statement. All SuperBASIC structures end with **END** something. That is consistent and sensible but the **NEXT** statement has a part to play as you will see in later chapters.

SELF TEST ON CHAPTER 6

You can score a maximum of 16 points from the following test. Check your score with the answers in the "Answers To Self Tests" section at the end of this Beginner's Guide.

1. Mention two difficulties which arise when the data needed for a program becomes numerous and you try to handle it without arrays (two points).

2. If, in an array, ten variables have the same name then how do you know which is which?

3. What must you do normally in a program, before you can use an array variable?

4. What is another word for the number which distinguishes a particular variable of an array from the other variables which share its name?

5. Can you think of two ideas in ordinary life which correspond to the concept of an array in programming?(two points)

6. In a REPeat loop, the process ends when some condition causes an EXIT statement to be executed. What causes the process in a FOR loop to terminate?

7. A REPeat loop needs a name so that you can EXIT to its END properly. A FOR loop also has a name, but what other function does a FOR loop name have?

8. What are the two phrases which are used to describe the variable which is also the name of a FOR loop?(two points)

9. The values of a loop variable change automatically as a FOR loop is executed. Name one possible important use of these values.

10. Which of the following do the long form of REPeat loops and the long form of FOR loops have in common? For each of the four items either say that both have it or which type of loop has it.

    1. An opening keyword or statement.
    2. A closing keyword or statement.
    3. A loop name.
    4. A loop variable or control variable. (four points)

PROBLEMS ON CHAPTER 6

1. Use a **FOR** loop to place one of four numbers 1,2,3,4 randomly in five array variables:

   card(1), card(2), card(3), card(4), card(5)

It does not matter if some of the four numbers are repeated. Use a second **FOR** loop to output the values of the five card variables.

2. Imagine that the four numbers 1,2,3,4 represent 'Hearts', 'Clubs', 'Diamonds', 'Spades'. What extra program lines would need to be inserted to get output in the form of these words instead of numbers?

3. Use a **FOR** loop to place five random numbers in the range 1 to 13 in an array of five variables:

   card(1), card(2) card(3), card(4) and *card(5)*

   Use a second **FOR** loop to output the values of the five card variables.

4. Imagine that the random numbers generated in problem 1 represent cards. Write down the extra statements that would cause the following output:

Number	**Output**
1	the word ‘Ace’
2 to 10	the actual number
11	the word ‘Jack’
12	the word ‘Queen’
13	the word ‘King’

CHAPTER 7 – SIMPLE PROCEDURES

If you were to try to write computer programs to solve complex problems you might find it difficult to keep track of things. A methodical problem solver therefore divides a large or complex job into smaller sections or **tasks**, and then divides these tasks again into smaller tasks, and so on until each can be be easily tackled.

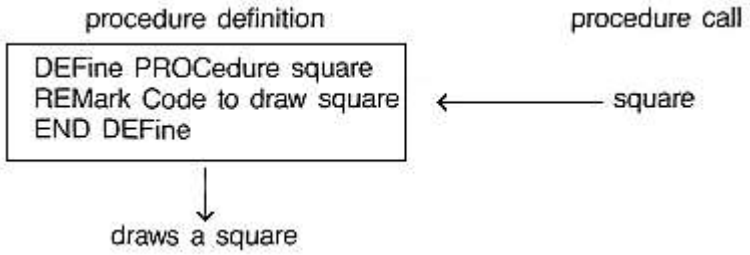
This is similar to the arrangement of complex human affairs. Successful government depends on a delegation of responsibility. The Prime Minister divides the work amongst ministers, who divide it further through the Civil Service until tasks can be done by individuals without further division. There are complicating features such as common services and interplay between the same and different levels, but the hierarchical structure is the dominant one.

A good programmer will also work in this way and a modern language like SuperBASIC which allows properly named, well defined procedures will be much more helpful than older versions which do not have such features.

The idea is that a separately named block of code should be written for a particular task. It doesn't matter where the block of code is in the program. If it is there somewhere,the use of its name will:

activate the code
return control to the point in the program immediately after that use.

If a procedure, *square*, draws a square the scheme is as shown below:



In practice the separate tasks within a job can be identified and named before the definition code is written. The name is all that is needed in calling the procedure so the main outline of the program can be written before all the tasks are defined.

Alternatively if it is preferred, the tasks can be written first and tested. If it works you can then forget the details and just remember the name and what the procedure does.

Example

The following example could quite easily be written without procedures but it shows they can be used in a reasonably simple context. Almost any task can be broken down in a similar fashion which means that you never have to worry about more than, say five to thirty lines at any one time. If you can write thirty-line programs well and handle procedures, then you have the capability to write three-hundred-line programs.

You can produce ready made buzz phrases for politicians or others who wish to give an impression of technological fluency without actually knowing anything. Store the following words in three arrays and then produce ten random buzz phrases.

adjec1$	adjec2$	noun$
Full	fifth-generation	systems
Systematic	knowledge-based	machines
Intelligent	compatible	computers

Controlled	cybernetic	feedback
Automated	user-friendly	transputers
Synchronised	parallel	micro-chips
Functional	learning	capability
Optional	adaptable	programming
Positive	modular	packages
Balanced	structured	databases
Integrated	logic-oriented	spreadsheets
Coordinated	file-oriented	word-processors
Sophisticated	Standardised	objectives

ANALYSIS

We will write a program to produce ten buzzword phrases. The stages of the program are:

1 Store the words in three string arrays.

2 Choose three random numbers which will be the subscripts of the array variables.

3 Print the phrase.

4 Repeat 2 and 3 ten times.

DESIGN - VARIABLES

We identify three arrays of which the first two will contain adjectives or words used as adjectives - describing words. The third array will hold the nouns. There are 13 words in each section and the longest word has 16 characters including a hyphen.

Array	**Purpose**
adjec1$(13,12)	first adjectives
adjec2$(13,16)	second adjectives
noun$(13,15)	nouns

DESIGN – PROCEDURES

We use three procedures to match the jobs identified.

store_data - stores the three sets of thirteen words.
get_random - gets three random numbers in range 1 to 13.
make_phrase - prints a phrase.

DESIGN - MAIN PROGRAM

This is very simple because the main work is done by the procedures.

Declare (**DIM**) the arrays
Store_data
FOR ten phrases
get_random
make_phrase
END

DESIGN - PROGRAM

```
100 REMark ************
110 REMark * Buzzword *
120 REMark ************
130 DIM adjec1$(13,12), adjec2$(13,16), noun$(13,15)
140 store_data
150 FOR phrase = 1 TO 10
160 get_random
170 make_phrase
180 END FOR phrase
190 REMark *************************
200 REMark * Procedure Definitions *
210 REMark *************************
220 DEFine PROCedure store_data
230 REMark *** procedure to store the buzzword data ***
240 RESTORE 420
250 FOR item = 1 TO 13
260READ adjec1$(item), adjec2$(item), noun$(item)
270 END FOR item
280 END DEFine
290 DEFine PROCedure get_random
300 REMark *** procedure to select the phrase ***
310 LET ad1 = RND(1 TO 13)
320 LET ad2 = RND(1 TO 13)
330 LET n = RND(1 TO 13)
340 END DEFine
350 DEFine PROCedure make_phrase
360 REMark *** procedure to print out the phrase ***
370 PRINT ! adjec!$(ad1) ! adjec2$(ad2) ! noun$(n)
380 END DEFine
390 REMark ****************
400 REMark * Program Data *
410 REMark ****************
420 DATA "Full", "fifth-generation", "systems"
430 DATA "Systematic", "knowledge-based", "machines"
440 DATA 'Intelligent","compatible", "computers"
450 DATA "Controlled", "cybernetic", "feedback"
460 DATA "Automated", "user-friendly", "transputers"
470 DATA "Synchronised", "parallel", "micro-chips"
480 DATA "Functional", "Learning", "capability'
490 DATA "Optional", "adaptable", "programming"
500 DATA "Positive" , "modular" , "packages"
510 DATA "Balanced" , "structured", "databases"
520 DATA "Integrated", "logic-oriented", "spreadsheets"
530 DATA "Coordinated", "file-oriented", "word-processors"
540 DATA "Sophisticated", "standardised", "objectives"


Automated fifth-generation capability
Functional learning packages
Full parallel objectives
Positive user-friendly spreadsheets
Intelligent file-oriented capability
Synchronised cybernetic transputers
Functional logic-oriented micro-chips
Positive parallel feedback
Balanced learning databases
Controlled cybernetic objectives
```

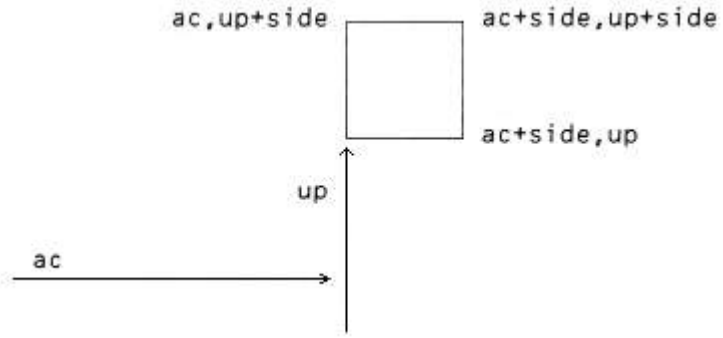
PASSING INFORMATION TO PROCEDURES

Suppose we wish to draw squares of various sizes and various colours in various positions on the scale graphics screen.

If we define a procedure, "square", to do this it will require four items of information:

length of one side
colour (colour code)
position (across and up)

The square's position is determined by giving two values, across and up, which fix the bottom left hand corner of the square as shown below.



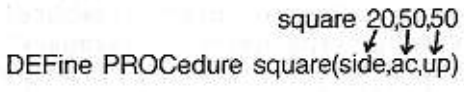
The colour of the square is easily fixed but the square itself uses the values of *side* and *ac* and *up* as follows.

```
200 DEFine PROCedure square(side,ac,up)
210   LINE ac,up TO ac+side,up
220   LINE TO ac+side,up+side
230   LINE TO ac,up+side TO ac,up
240 END DEFine
```

In order to make this procedure work values of *side*, *ac* and *up* must be provided. These values are provided when the procedure is called. For example you could add the following main program to get one green square of side 20.

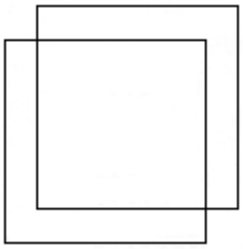
```
100 PAPER 7:CLS
110 INK 4
120 square 20,50,50
```

The numbers 20,50,50 are called parameters and they are passed to the variables named in the procedure definition thus:



The numbers 20,50,50 are called **actual parameters**. They are numbers in this case but they could be variables or expressions. The variables *side, ac, up* are called **formal parameters**. They must be variables because they 'receive' values.

A more interesting main program uses the same procedure to create a random pattern of coloured pairs of squares. Each pair of squares is obtained by offsetting the second one across and up by one-fifth of the side length thus:



Assuming that the procedure **square** is still present at line 200 then the following program will have the classical effect.

```
100 REMark Squares Pattern
110 PAPER 7 : CLS
120 FOR pair = 1 TO 20
130   INK RND(5)
140   LET side = RND(10 TO 20)
150   LET ac = RND(50) : up = RND(70)
160   square side,ac,up
170   LET ac=ac+side/5 : up = up+side/5
180   square side,ac,up
190 END FOR pair
```

The advantages of procedures are:

1. You can use the same code more than once in the same program or in others.

2. You can break down a task into sub-tasks and write procedures for each sub-task. This helps the analysis and design.

3. Procedures can be tested separately. This helps the testing and debugging.

4. Meaningful procedure names and clearly defined beginnings and ends help to make a program readable.

When you get used to properly named procedures with good parameter facilities, you should find that your problem-solving and programming powers are greatly enhanced.

SELF TEST ON CHAPTER 7

You can score a maximum of 14 points from the following test. Check your score with the "Answers To Self Tests" section at the back of this Beginner's Guide.

1. How do we normally tackle the problem of great size and complexity in human affairs?

2. How can this principle be applied in programming?

3. What are the two most obvious features of a simple procedure definition? (two points)

4. What are the two main effects of using a procedure name to 'call' the procedure? (two points)

5. What is the advantage of using procedure names in a main program before the procedure definitions are written?

6. What is the advantage of writing a procedure definition before using its name in a main program?

7. How can the use of procedures help a 'thirty-line-programmer' to write much bigger programs?

8. Some programs use more memory in defining procedures, but in what circumstances do procedures save memory space?

9. Name two ways by which information can be passed from main program to a procedure. (two points)

10. What is an actual parameter?

11. What is a formal parameter?

PROBLEMS ON CHAPTER 7

1. Write a procedure which outputs one of the four suits: 'Hearts', 'Clubs: 'Diamonds' or 'Spades'. Call the procedure five times to get five random suits.

2. Write another program for problem 1 using a number in the range 1 to 4 as a parameter to determine the output word. If you have already done this, then try writing the program without parameters.

3. Write a procedure which will output the value of a card that is a number in the range 2 to 10 or one of the words 'Ace', 'Jack' 'Queen', 'King'.

4. Write a program which calls this procedure five times so that five random values are output.

5. Write the program of problem 3 again using a number in the range 1 to 13 as a parameter to be passed to the procedure. If this was the method you used first time, then try writing the program without parameters.

6. Write the most elegant program you can, using procedures, to output four hands of five cards each. Do not worry about duplicate cards. You can take elegance to mean an appropriate mixture of readability shortness and efficiency. Different people and/or different circumstances will place different importance on these three qualities which sometimes work against each other.

CHAPTER 8 – FROM BASIC TO SUPERBASIC

If you are familiar with one of the earlier versions of BASIC you may find it possible to omit the first seven chapters and use this chapter instead as a bridge between what you know already and the remaining chapters. If you do this and still find areas of difficulty. it may be helpful to backtrack a little into some of the earlier chapters.

If you have worked through the earlier chapters this one should be easy reading. You may find that, as well as introducing some new ideas, it gives an interesting slant on the way BASIC is developing. Apart from its program structuring facilities SuperBASIC also pushes forward the frontiers of good screen presentation, editing, operating facilities and graphics. In short it is a combination of user-friendliness and computing power which has not existed before.

So, when you make the transition from BASIC to SuperBASIC you are moving not only to a more powerful, more helpful language, you are also moving into a remarkably advanced computing environment.

We will now discuss some of the main features of SuperBASIC and some of the features which distinguish it from other BASICs.

ALPHABETIC COMPARISONS

The usual simple arithmetic comparisons are possible. You can write:

```
LET pet1$ = "CAT"
LET pet2$ = "DOG"
IF pet1$ < pet2$ THEN PRINT "Meow"
```

The output will be Meow because in this context the symbol **<** means:

earlier (nearer to A in the alphabet)

SuperBASIC makes comparisons sensible. For example you would expect:

'cat' to come before 'DOG'

and

'ERD98L' to come before 'ERD746L'

A simplistic approach, blindly using internal character coding, would give the 'wrong' result in both the above cases but try the following program which finds the 'earliest' of two character strings.

```
100 INPUT item1$, item2$
110 IF item1$ < item2$ THEN PRINT item1$
120 IF item1$ = item2$ THEN PRINT "Equal"
130 IF item1$ > item2$ THEN PRINT item2$
```

INPUT		OUTPUT
cat	dog	cat
cat	DOG	cat
ERD98L	ERD746L	ERD98L
ABC	abc	ABC

The *Concept Reference Guide* section will give full details about the way comparisons of strings are made in SuperBASIC.

VARIABLES AND NAMES - IDENTIFIERS

Most BASICs have numeric and string variables. As in other BASICs the distinguishing feature of a string variable name in SuperBASIC is the dollar sign on the end. Thus:

numeric:	count	string:	*word$*
	sum		*high_st$*
	total		*day_of_week$*

You may not have met such meaningful variable names before though some of the more recent BASICs do allow them. The rules for identifiers in SuperBASIC are given in the *Concept Reference Guide*. The maximum length of an identifier is 255 characters. Your choice of identifiers is a personal one. Sometimes the longer ones are more helpful in conveying to the human reader what a program should do. But they have to be typed and, as in ordinary English, *spade* is more sensible than *horticultural earth-turning implement*. Shorter words are preferred if they convey the meaning but very short words or single letters should be used sparingly. Variable names like *X, Z, P3, Q2* introduce a level of abstraction which most people find unhelpful.

INTEGER VARIABLES

SuperBASIC allows **integer** variables which take only whole-number values. We distinguish these with a percentage sign thus:

```
count%
number%
nearest_pound%
```

There are now two kinds of numeric variable. We call the other type, which can take whole or fractional values, **floating point**. Thus you can write:

```
LET price = 9
LET cost = 7.31
LET count% = 13
```

But if you write:

```
LET count% = 5.43
```

the value of *count%* will become 5. On the other hand:

```
LET count% = 5.73
```

will cause the value of *count%* to be 6. You can see that SuperBASIC does the best it can, rounding off to the nearest whole number.

COERCION

The principle of always trying to be intelligently helpful,rather than give an error message or do something obviously unwanted, is carried further. For example, if a string variable **mark$** has the value

```
'64'
```

then:

```
LET score = mark$
```

will produce a numeric value of 64 for score. Other versions of BASIC would be likely to halt and say something like:

```
'Type mis-match'
or 'Nonsense in BASIC'
```

If the string cannot be converted then an error is reported.

LOGICAL VARIABLES AND SIMPLE PROCEDURES

There is one other type of variable in SuperBASIC, or rather the SuperBASIC system makes it seem so. Consider the SuperBASIC statement:

```
IF windy THEN fly_kite
```

In other BASICs you might write:

```
IF w=1 THEN GOSUB 300
```

In this case *w=1* is a condition or logical expression which is either true or false. If it is true then a subroutine starting at line 300 would be executed. This subroutine may deal with kite flying but you cannot tell from the above line. A careful programmer would write:

```
IF w=1 THEN GOSUB 300 : REM fly_kite
```

to make it more readable. But the SuperBASIC statement is readable as it stands. The identifier *windy* is interpreted as true or false though it is actually a floating point variable. A value of 1 or any non-zero value is taken as *true*. Zero is taken as false. Thus the single word, *windy,* has the same effect as a condition of logical expression.

The other word, *fly_kite*, is a procedure. It does a job similar to, but rather better than, **GOSUB 300**.

The following program will convey the idea of logical variables and the simplest type of named procedure.

```
100 INPUT windy
110 IF windy THEN fly_kite
120 IF NOT windy THEN tidy_shed
130 DEFine PROCedure fly_kite
140 PRINT "See it in the air."
150 END DEFine
160 DEFine PROCedure tidy_shed
170 PRINT "Sort out rubbish."
180 END DEFine
```

INPUT	OUTPUT
0	Sort out rubbish
1	See it in the air
2	See it in the air
-2	See it in the air

You can see that only zero is taken as meaning false. You would not normally write procedures with only one action statement, but the program illustrates the idea and syntax in a very simple context. More is said about procedures later in this chapter.

LET STATEMENTS

In SuperBASIC **LET** is optional but we use it in this manual so that there will be less chance of confusion caused by the two possible uses of **=**. The meanings of **=** in:

```
LET count = 3
```

and in

```
IF count = 3 THEN EXIT
```

are different and the **LET** helps to emphasise this. However if there are two or a few **LET** statements doing some simple job such as setting initial values, an exception may be made.

For example:

```
100 LET first = 0
110 LET second = 0
120 LET third = 0
```

may be re-written as

```
100 LET first = 0 : second = 0 : third = 0
```

without loss of clarity or style. It is also consistent with the general concept of allowing short forms of other constructions where they are used in simple ways.

The colon : is a valid statement terminator and may be used with other statements besides **LET**.

THE BASIC SCREEN

In a later chapter we will explain how other graphics facilities, such as drawing circles, can be handled but here we outline the pixel-oriented features. There are two modes which may be activated by any of the following:

Low resolution **8 Colour Mode** **256 pixels across, 256 down**	**MODE 256** **MODE 8**
High resolution **4 Colour Mode** **512 pixels across, 256 down**	**MODE 512** **MODE 4**

In both modes pixels are addressed by the range of numbers:

0 - 511 across

and 0 - 255 down

Since mode 8 has only half the number of pixels across the screen as mode 4, mode 8 pixels are twice as wide as mode 4 pixels and so in mode 8 each pixel can be specified by two coordinates. For example:

0 or 1 2 or 3 510 or 511

It also means that you use the same range of numbers for addressing pixels irrespective of the mode. Always think 0-511 across and 0-255 down.

If you are using a television then not all the pixels may be visible.

The colours available are:

MODE 256	**Code**	**MODE 512**
Black	0	Black
Blue	1	
Red	2	Red
Magenta	3	
Green	4	Green
Cyan	5	

Yellow	6	white
white	7	

You may find the following mnemonic helpful in remembering the codes:

Bonny Babies Really Make Good Children, You Wonder

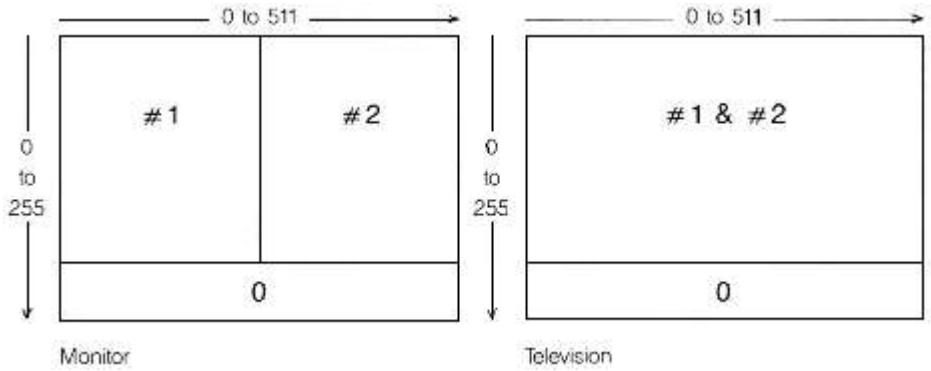
In the *high resolution* mode each colour can be selected by one of two codes. You will see later how a startling range of colour and stipple (texture) effects can be produced if you have a good quality colour monitor.

Some of the screen presentation keywords are as follows:

INK *colour*	foreground colour
BORDER *width, colour*	draw border at edge of screen or window
PAPER *colour*	background colour
BLOCK *width, height, across, down, colour*	colour a rectangle which has its top left hand corner at position across, down

SCREEN ORGANISATION

When you switch on your QL the screen display is split into three areas called *windows* as shown below. Note that in order to fit these windows into the area covered by a television screen, some pixels around the border are not used in Television mode.



The windows are identified by #0, #1 and #2 so that you can relate various effects to particular windows. For example:

```
CLS
```

will clear window #1 (the system chooses) so if you want the left hand area cleared you must type:

```
CLS #2
```

If you want a different paper (background colour) type for green:

```
PAPER 4 : CLS
```

or

```
PAPER #2,4 : CLS #2
```

If you want to clear window #2 to the background colour green.

The numbers #0, #1 and #2 are called *channel numbers*. In this particular case they enable you to direct certain effects to the window of your choice. You will discover later that channel numbers have many other uses but for the moment note that all of the following statements may have a channel number. The third column shows the default channel - the one chosen by the system if you do not specify one.

Note that windows may overlap. If you use a TV screen the system automatically overlaps windows #1 and #2 so that more character positions per line are available for program listings.

KEYWORD	EFFECT	DEFAULT
AT	Character position	#1
BLOCK	Draws block	#1
BORDER	Draw border	#1
CLS	Clear screen	#1
CSIZE	Character size	#1
CURSOR	Position cursor	#1
FLASH	Causes/cancels flashing	#1
INK	Foreground colour	#1
OVER	Effect of printing and graphics	#1
PAN	Moves screen sideways	#1
PAPER	Background colour	#1
RECOL	Changes colour	#1
SCROLL	Moves screen vertically	#1
STRIP	Background for printing	#1
UNDER	Underlines	#1
WINDOW	Changes existing window	#1
LIST	Lists program	#2
DIR	Lists directory	#1
PRINT	Prints characters	#1
INPUT	Takes keyboard input	#1

Statements or direct commands appear in window #0.

For more details about the syntax or use of these keywords see other parts of the manual.

RECTANGLES AND LINES

The program below draws a green rectangle in 256 mode on red paper with a yellow border one pixel wide. The rectangle has its top left corner at pixel co-ordinates 100,100 (*see QL Concepts*). Its width is 80 units across (40 pixels) and its height is 20 units down (20 pixels).

```
100 REMark Rectangle
110 MODE 256
120 BORDER 1,6
130 PAPER 2 : CLS
140 BLOCK 80,20,100,100,4
```

You have to be a bit careful in mode 256 because across values range from 0 to 511 even though there are only 256 pixels. We cannot say that the block produced by the above program is 80 pixels wide so we say 80 units.

INPUT AND OUTPUT

SuperBASIC has the usual **LET**, **INPUT**, **READ** and **DATA** statements for input. The **PRINT** statement handles most text output in the usual way with the separators:

, tabulates output

; just separates - no formatting effect
\ forces new line
! normally provides a space but not at the start of line. If an item will not fit at the end of a line it performs a new line operation.
TO Allows tabulation to a designated column position.

You will be familiar with two types of repetitive loop exemplified as follows:

(a) Simulate 6 throws of an ordinary six-sided die

```
100 FOR throw = 1 TO 6
110 PRINT RND(1 TO 6)
120 NEXT throw
```

(b) Simulate throws of a die until a six appears.

```
100 die = RND(1 TO 6)
110 PRINT die
120 IF die <> 6 THEN GOTO 10
```

Both of these programs will work in SuperBASIC but we recommend the following instead. They do exactly the same jobs. Although program (b) is a little more complex there are good reasons for preferring it.

Program (a)

```
100 FOR throw = 1 TO 6
110 PRINT RND(1 TO 6)
120 END FOR throw
```

Program (b)

```
100 REPeat throws
110 die = RND(1 TO 6)
120 PRINT die
130 IF die = 6 THEN EXIT throws
140 END REPeat throws
```

It is logical to provide a structure for a loop which terminates on a condition (**REPeat** loops) as well as those which are controlled by a count.

The fundamental REPeat structure is:

REPeat identifier
  statements
END REPeat identifier

The EXIT statement can be placed anywhere in the structure but it must be followed by an identifier to tell SuperBASIC which loop to exit; for example:

EXIT throws

would transfer control to the statement after

END REPeat throws.

This may seem like a using a sledgehammer to crack the nut of the simple problem illustrated. However the **REPeat** structure is very powerful. It will take you a long way.

If you know other languages you may see that it will do the jobs of both **REPEAT** and **WHILE** structures and also cope with other more awkward, situations.
The SuperBASIC **REPeat** loop is named so that a correct clear exit is made. The **FOR** loop, like all SuperBASIC structures, ends with **END**, and its name is given for reasons which will become clear later.

You will also see later how these loop structures can be used in simple or complex situations to match exactly what you need to do. We will mention only three more features of loops at this stage. They will be familiar if you are an experienced user of BASIC.

The increment of the control variable of a **FOR** loop is normally 1 but you can make it other values by using the STEP keyword. As the examples show.

Example (i).

```
100 FOR even = 2 TO 10 STEP 2
110 PRINT ! even !
120 END FOR even
```

output is `2 4 6 8 10`

Example (ii).

```
100 FOR backwards = 9 TO 1 STEP -1
110 PRINT ! backwards !
120 END FOR backwards
```

output is `9 8 7 6 5 4 3 2 1`

The second feature is that loops can be nested. You may be familiar with nested **FOR** loops. For example the following program outputs four rows of ten crosses.

```
100 REMark Crosses
110 FOR row = 1 TO 4
120 PRINT 'Row number' ! row
130  FOR cross = 1 TO 10
140  PRINT ! 'X' !
150 END FOR cross
160 PRINT
170 PRINT \ 'End of row number' ! row
180 END FOR row
```

output is:

```
Row number 1
X X X X X X X X X X
End of row number 1
Row number 2
X X X X X X X X X X
End of row number 2
Row number 3
X X X X X X X X X X
End of row number 3
Row number 4
X X X X X X X X X X
End of row number 3
```

A big advantage of SuperBASIC is that it has structures for all purposes, not just **FOR** loops, and they can all be nested one inside the other reflecting the needs of a task. We can put a **REPeat** loop in a

FOR loop. The program below produces scores of two dice in each row until a seven occurs, instead of crosses.

```
100 REMark Dice rows
110 FOR row = 1 TO 4
120 PRINT 'Row number '! row
130 REPeat throws
140   LET die1 = RND(1 TO 6)
150   LET die2 = RND(1 TO 6)
160   LET score = die1 + die2
170   PRINT ! score !
180   IF score = 7 THEN EXIT throws
190 END REPeat throws
200 PRINT \'End of row '! row
210 END FOR row
```

sample output:

```
Row number 1
8 11 6 3 7
End of row number 1
Row number 2
4 6 2 9 4 5 12 7
End of row number 2
Row number 3
7
End of row number 3
Row number 4
6 2 4 9 9 7
End of row number 4
```

The third feature of loops in SuperBASIC allows more flexibility in providing the range of values in a **FOR** loop. The following program illustrates this by printing all the divisible numbers from 1 to 20. (A divisible number is divisible evenly by a number other than itself or 1.)

```
100 REMark Divisible numbers
110 FOR num = 4,6,8, TO 10,12,14 TO 16,18, 20
120   PRINT ! num !
130 END FOR num
```

More will be said about handling repetition in a later chapter but the features described above will handle all but a few uncommon or advanced situations.

DECISION MAKING

You will have noticed the simple type of decision:

```
IF die = 6 THEN EXIT throws
```

This is available in most BASICs but SuperBASIC offers extensions of this structure and a completely new one for handling situations with more than two alternative courses of action.

However, you may find the following long forms of **IF..THEN** useful. They should explain themselves.

(i)

```
100 REMark Long form IF. ..END IF
110 LET sunny = RND(0 TO 1)
120 IF sunny THEN
130 PRINT 'Wear sunglasses'
140 PRINT 'Go for walk'
```

```
150 END IF
```

(ii)

```
100 REMark Long form IF...ELSE...END IF
110 LET sunny = RND(0 TO 1)
120 IF sunny THEN
130 PRINT 'Wear sunglasses'
140 PRINT 'Go for walk'
150 ELSE
160 PRINT 'Wear coat'
170 PRINT 'Go to cinema'
180 END IF
```

The separator **THEN**, is optional in long forms or it can be replaced by a colon in short forms. The long decision structures have the same status as loops. You can nest them or put other structures into them. When a single variable appears where you expect a condition the value zero will be taken as false and other values as true.

SUBROUTINES AND PROCEDURES

Most BASICs have a **GOSUB** statement which may be used to activate particular blocks of code called subroutines. The **GOSUB** statement is unsatisfactory in a number of ways and SuperBASIC offers properly named procedures with some very useful features.

Consider the following programs both of which draw a green 'square' of side length 50 pixel screen units at a position 200 across 100 down on a red background.

(a) Using **GOSUB**

```
100 LET colour = 4 : background = 2
110 LET across = 20
120 LET down = 100
130 LET side = 50
140 GOSUB 170
150 PRINT 'END'
160 STOP
170 REMark Subroutine to draw square
180 PAPER background : CLS
190 BLOCK side, side, across, down, colour
200 RETurn
```

(b) Using a procedure with parameters

```
100 square 4, 50, 20, 100, 2
110 PRINT 'END'
120 DEFine PROCedure square(colour,side,across,down,background)
130 PAPER background : CLS
140 BLOCK side, side, across, down, colour
150 END DEFine
```

In the firs t program the values of *colour, across, down, side* are fixed by **LET** statements before the **GOSUB** statement activates lines 180 and 190 Control is then sent back by the **RETURN** statement.

In the second program the values are given in the first line as parameters in the procedure call, *square*, which activates the procedure and at the same time provides the values it needs.

In its simplest form a procedure has no parameters. It merely separates a particular piece of code, though even in this simpler use the procedure has the advantage over **GOSUB** because it is properly named and properly isolated into a self contained unit.

The power and simplifying effects of procedures are more obvious as programs get larger. What procedures do as programs get larger is not so much make programming easier as prevent it from getting harder with increasing program size. The above example just illustrates the way they work in a simple context.

Examples

The following examples indicate the range of vocabulary and syntax of SuperBASIC which has been covered in this and earlier chapters, and will form a foundation on which the second part of this manual will build.

The letters of a palindrome are given as single items in DATA statements. The terminating item is an asterisk and you assume no knowledge of the number of letters in the palindrome. READ the letters into an array and print them backwards. Some palindromes such as "MADAM I'M ADAM" only work if spaces and punctuation are ignored. The one used here works properly.

```
100 REMark Palindromes
110 DIM text$(30)
120 LET text$ = FILL$ (' ',30)
130 LET count = 30
140 REPeat get_letters
150   READ character$
160   IF character$ = '*' THEN EXIT get_letters
170   LET count = count-1
180   LET text$(count) = character$
190 END REPeat get_letters
200 PRINT text$
210 DATA 'A','B','L','E','W','A','S','I','E','R'
220 DATA 'E','I','S','A','W','E','L','B','A','*'
```

The following program accepts as input numbers in the range 1 to 3999 and converts them into the equivalent In Roman numerals It does not generate the most elegant form. It produces IIII rather than IV.

```
100 REMark Roman numbers
110 INPUT number
120 RESTORE 210
130 FOR type = 1 TO 7
140   READ letter$, value
150   REPeat output
160     IF number < value : EXIT output
170     PRINT letter$;
180     LET number = number - value
190   END REPeat output
200 END FOR type
210 DATA 'M',1000,'D',500,'C',100,'L',50,'X',10,'V',5,'I',1
```

You should study the above examples carefully using dry runs if necessary until you are sure that you understand them.

CONCLUSION

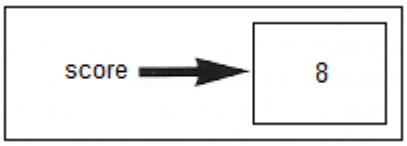
In SuperBASIC full structuring features are provided so that program elements either follow in sequence or fit into one another neatly. All structures must be identified to the system and named. There are many unifying and simplifying features and many extra facilities.

Most of these are explained and illustrated in the remaining chapters of this manual, which should be easier to read than the Keyword and Concept Reference sections. However, it is easier to read because it does not give every technical detail and exhaust every topic which it treats. There may, therefore, be a few occasions when you need to consult the reference sections. On the other hand some major advances are discussed in the following chapters. Few readers will need to use all of them and you may find it helpful to omit certain parts, at least on first reading.

CHAPTER 9 - DATA TYPES VARIABLES AND IDENTIFIERS

You will have noticed that a program (a sequence of statements) usually gets some data to work on (input) and produces some kind of results (output). You will also have understood that there are internal arrangements for storing this data. In order to avoid unnecessary technical explanations we have suggested that you imagine pigeon holes and that you choose meaningful names for the pigeon holes. For example, if it is necessary to store a number which represents the score from simulated dice-throws you imagine a pigeon hole named score which might contain a number such as 8.

Internally the pigeon holes are numbered and the system maintains a dictionary which connects particular names with particular numbered pigeon holes. We say that the name, score, points to its particular pigeon-hole (by means of the internal dictionary).



The whole arrangement is called a **variable**.

What you see is the word score. We say that this word, *score* is an identifier It is what we see and it identifies the concept we need, in this case the result, 8, of throwing a pair of dice. Because the identifier is what we see it becomes the thing we talk or write or think about. We write about score and its value at any particular moment.

There are four simple data types called floating point, integer string and logical and these are explained below We talk about data types rather than variable types because data can occur on its own, for example 3.4 or 'Blue hat' as the value of a variable. But if you understand the different types of variables, you must also understand the different types of data.

IDENTIFIERS AND VARIABLES

1. A SuperBASIC identifier must begin with a letter and is a sequence of:

   upper or lower case letters
   digits or underscore

2. An identifier may be up to 255 characters in length so there is no effective limit in practice.

3. An identifier cannot be the same as a keyword of SuperBASIC.

4. An integer variable name is an identifier with % on the end.

5. A string variable name is an identifier with $ on the end.

6. No other identifiers must use the symbols % and $.

7. An identifier should usually be chosen so that it means something to a human reader but for SuperBASIC it does not have any particular meaning other than that it identifies certain things.

FLOATING POINT VARIABLES

Examples of the use of floating point variables are:

```
100 LET days = 24
110 LET sales = 3649.84
120 LET sales_per_day = sales/days
130 PRINT sales_per_day
```

The value of a floating point variable may be anything in the range:
$\pm 10^{-615}$ to $\pm 10^{+615}$ with 8 significant figures.

Suppose in the above program sales were, exceptionally only 3p. Change line 110 to:

```
110 LET sales = 0.03
```

This system will change this to:

```
110 LET sales = 3E-2
```

To interpret this, start with 3 or 3.0 and move the decimal point -2 places, i.e. two places left. This shows that:

3E-2 is the same as 0.03

After running the program, the average daily sales are:

1.25E-3 which is the same as 0.00125

Numbers with an E are said to be in exponent form:

(mantissa) E (exponent) = (mantissa) x 10 to the power (exponent)

INTEGER VARIABLES

Integer variables can have only whole number values in the range -32678 to 32768. The following are examples of valid integer variable names which must end with %.

```
LET count% = 10
LET six_tally% = RND(10)
LET number_3% = 3
```

The only disadvantage of integer variables, when whole numbers are required, is the slightly misleading % symbol on the end of the identifier. It has nothing to do with the concept of percentage. It is just a convenient symbol tagged on to show that the variable is an integer.

NUMERIC FUNCTIONS

Using a function is a bit like making an omelette. You put in an egg which is processed according to certain rules (the recipe) and get out an omelette. For example the function **INT** takes any number as input and outputs the whole number part. Anything which is input to a function is called a parameter or argument. **INT** is a function which gives the integer part of an expression. You may write:

```
PRINT INT(5.6)
```

and 5 would be the output. We say that 5.6 is the parameter and the function returns the value 5. A function may have more than one parameter. You have already met:

```
RND(1 TO 6)
```

which is a function with two parameters. But functions always return exactly one value. This must be so because you can put functions into expressions. For example:

```
PRINT 2 * INT(5.6)
```

would produce the output 10. It is an important property of functions that you can use them in expressions. It follows that they must return a single value which is then used in the expression. **INT** and **RND** are system functions: they come with the system, but later you will see how to write your own.

The following examples show common uses of the **INT** function.

```
100 REMark Rounding
110 INPUT decimal
120 PRINT INT(decimal + 0.5)
```

In the example you input a decimal fraction and the output is rounded. Thus 4.7 would become 5 but 4.3 would become 4.

You can achieve the same result using an integer variable and coercion.

Trigonometrical functions will be dealt with in a later section but other common numeric functions are given in the list below.

Function	Effect	Examples	Returned values
ABS	Absolute or unsigned value	ABS(7) ABS(-4.3)	7 4.3
INT	Integer part of a floating point number	INT(2.4) INT(0.4) INT(-2.7)	2 0 -3
SQRT	Square root	SQRT(2) SQRT(16) SQRT(2.6)	1.414214 4 1.612452

There is a way of computing square roots which is easy to understand. To compute the square root of 8 first make a guess. It doesn't matter how bad the guess maybe. Suppose you simply take half of 8 as the first guess which is 4.

Because 4 is greater than the square root of 8 then 8/4 must be less than it. The reverse is also true. If you had guessed 2 which is less than the square root then 8/2 must be greater than it.

It follows that if we take any guess and computer number/guess we have two numbers, one too small and one too big. We take the average of these numbers as our next approximation and thus get closer to the correct answer.

We repeat this process until successive approximations are so close as to make little difference:

```
100 REMark Square Roots
110 LET number = 8
120 LET approx = number/2
130 REPeat root
140   LET newval = (approx + number/approx)/2
150   IF newval == approx THE EXIT root
160   LET approx = newval
170 END REPeat root
180 PRINT 'Square root of' ! number ! 'is' ! newval
```

sample output:

```
Square root of 8 is 2.828427
```

Notice that the conditional **EXIT** from the loop must be in the middle. The traditional structures do not cope with this situation as well as SuperBASIC does. The == sign in line 150 means "approximately equal to", that is, equal to within .0000001 of the values being compared.

NUMERIC OPERATIONS

SuperBASIC allows the usual mathematical operations. You may notice that they are like functions with exactly two operands each. It is also conventional in these cases to put an operand on each side of the symbol. Sometimes the operation is denoted by a familiar symbol such as + or *. Sometimes the operation is denoted by a keyword like **DIV** or **MOD** but there is no real difference. Numeric operations have an order of priority. For example, the result of:

```
PRINT 7 + 3*2
```

is 13 because the multiplication has a higher priority. However:

```
PRINT (7+3)*2
```

will output 20, because brackets over-ride the usual priority. As you will see later so many things can be done with SuperBASIC expressions that a full statement about priority cannot be made at this stage (see the *Concept Reference Guide* if you wish) but the operations we now deal with have the following order of priority:

highest - raising to a power
multiplication and division (including DIV, MOD)
lowest - add and subtract

The symbols + and - are also used with only one operand which simply denotes positive or negative. Symbols used in this way have the highest priority of all and can only be over-ridden by the use of brackets.

Finally if two symbols have equal priority the leftmost operation is performed first so that:

```
PRINT 7-2 + 5
```

will cause the subtraction before the addition. This might be important if you should ever deal with very large or very small numbers.

Operation	Symbol	Examples	Results	Note
Add	+	7+6.6	13.6	
Subtract	-	7-6.6	0.4	
Multiply	*	3*2.1	6.3	
		2.1*(-3)	-6.3	
Divide	/	7/2	3.5	Do not divide by zero
		-17/5	-3.4	
Raise to power	^	4^1.5	8	
Integer divide	DIV	-8 DIV 2	-4	Integers only
		7 DIV 2	3	Do not divide by zero
Modulus	MOD	13 MOD 5	3	
		21 MOD 7	0	
		-17 MOD 8	7	

Modulus returns the remainder part of a division. Any attempt to divide by zero will generate an error and terminate program execution.

NUMERIC EXPRESSIONS

Strictly speaking, a numeric expression is an expression which evaluates to a number and there are more possibilities than we need to discuss here. SuperBASIC allows you to do complex things if you want to but it also allows you to do simple things in simple ways. In this section we concentrate on those usual straightforward uses of mathematical features.

Basically numeric expressions in SuperBASIC are the same as those of mathematics but you must put the whole expression in the form of a sequence.

$$\frac{5+3}{6-4}$$

becomes in SuperBASIC (or other BASIC):

(5 + 3)/(6 - 4)

In secondary school algebra there is an expression for one solution of a quadratic equation:

$ax^2 + bx + c = 0$

One solution in mathematical notation is:

$$x = \frac{-b + \sqrt{b2 - 4ac}}{2a}$$

If we start with the equation:

$2x^2 - 3x + 1 = 0$

Example 1

The following program will find one solution.

```
100 READ a,b,c
110 PRINT 'Root is' ! (-b+SQRT(b^2 - 4*a*c))/(2*a)
120 DATA 2,-3,1
```

Example 2

In problems which need to simulate the dealing of cards you can make cards correspond to the numbers 1 to 52 as follows:

1 to 13
14 to 26
27 to 39
40 to 52
Ace, two.....king of hearts
Ace, two.....king of clubs
Ace, two.....king of diamonds
Ace, two.....king of spades

A particular card can be identified as follows:

```
100 REM Card identification
110 LET card = 23
120 LET suit = (card-1) DIV 13
130 LET value = card MOD 13
140 IF value = 0 THEN LET value = 13
150 IF value = 1 THEN PRINT "Ace of ";
160 IF value >= 2 AND value <= 10 THEN PRINT value ! "of ";
170 IF value = 11 THEN PRINT "Jack of ";
180 IF value = 12 THEN PRINT "Queen of ";
190 IF value = 13 THEN PRINT "King of ";
200 IF suit = 0 THEN PRINT "hearts"
```

```
210 IF suit = 1 THEN PRINT "clubs"
220 IF suit = 2 THEN PRINT "diamonds"
230 IF suit = 3 THEN PRINT "spades"
```

There are new ideas in this program. They are in line 160. The meaning is clearly that the number is actually printed only if two logical statements are true. These are:

value is greater than or equal to 2 AND value is less than or equal to 10

Cards outside this range are either aces or 'court cards' and must be treated differently

Note also the use of ! in the PRINT statement to provide a space and ; to ensure that output continues on the same line.

There are two groups of mathematical functions which we have not discussed here. They are the trigonometric and logarithmic. You may need the former in organising screen displays. Types of functions are also fully defined in the reference section.

LOGICAL VARIABLES

Strictly speaking, SuperBASIC does not allow logical variables but it allows you to use other variables as logical ones. For example you can run the following program:

```
100 REMark Logical Variable
110 LET hungry = 1
120 IF hungry THEN PRINT "Have a bun"
```

You expect a logical expression in line 120 but the numeric variable, hungry is there on its own. The system interprets the value, 1, of hungry as true and the output is:

```
Have a bun
```

If line 110 read:

```
LET hungry = 0
```

there would be no output. The system interprets zero as false and all other values as true. That is useful but you can disguise the numeric quality of hungry by writing:

```
100 REMark Logical Variable
110 LET true = 1 : false = 0
120 LET hungry = true
130 IF hungry THEN PRINT "Have a bun"
```

STRING VARIABLES

There is much to be said about handling strings and string variables and this is left to a separate chapter.

PROBLEMS ON CHAPTER 9

1. A rich oil dealer gambles by tossing a coin in the following way. If it comes down heads he gets 1. If it comes down tails he throws again but the possible reward is doubled. This is repeated so that the rewards are as shown.

THROW	1	2	3	4	5	6	7
REWARDS	1	2	4	8	16	32	64

   By simulating the game try to decide what would be a fair initial payment for each such game:

(a) if the player is limited to a maximum of seven throws per game.
(b) if there is no maximum number of throws

2. Bill and Ben agree to gamble as follows. At a given signal each divides his money into two halves and passes one half to the other player. Each then divides his new total and passes half to the other. Show what happens as the game proceeds if Bill starts with 16p and Ben starts with 64p.

3. What happens if the game is changed so that each hands over an amount equal to half of what the other possesses?

4. Write a program which forms random three letter words chosen from A,B,C,D and prints them until ' BAD ' appears.

5. Modify the last program so that it terminates when any real three letter word appears.

CHAPTER 10 – LOGIC

If you have read previous chapters you will probably agree that repetition, decision making and breaking tasks into sub-tasks are major concepts in problem analysis, program design and encoding programs. Two of these concepts, repetition and decision making, need logical expressions such as those in the following program lines:

```
IF score = 7 THEN EXIT throws
IF suit = 3 THEN PRINT "spades"
```

The first enables **EXIT** from a **REPeat** loop. The second is simply a decision to do something or not. A mathematical expression evaluates to one of millions of possible numeric values. Similarly a string expression can evaluate to millions of possible strings of characters. You may find it strange that logical expressions, for which great importance is claimed, can evaluate to one of only two possible values: *true* or *false*.

In the case of

```
score = 7
```

this is obviously correct. Either score equals 7 or it doesn't! The expression must be true or false - assuming that it's not meaningless. It may be that you do not know the value at some time, but that will be put right in due course.

You have to be a bit more careful of expressions involving words such as *OR, AND, NOT* but they are well worth investigating - indeed, they are essential to good programming. They will become even more important with the trend towards other kinds of languages based more on precise descriptions of what you require rather than what the computer must do.

AND

The word **AND** in SuperBASIC is like the word 'and' in ordinary English. Consider the following program.

```
100 REMark AND
110 PRINT "Enter two values" \ "1 for TRUE or 0 for FALSE"
120 INPUT raining, hole_in_roof
130 IF raining AND hole_in_roof THEN PRINT "Get wet"
```

As in real life, you will only get wet if it is raining and there is a hole in the roof. If one (or both) of the simple logical variables

raining
hole_in_roof

is false then the compound logical expression

raining **AND** *hole_in_roof*

is also false. It takes two true values to make the whole expression true. This can be seen from the rules below. Only when the compound expression is true do you get wet.

raining	hole_in_roof	raining and hole_in_roof	effect
FALSE	FALSE	FALSE	DRY
FALSE	TRUE	FALSE	DRY
TRUE	FALSE	FALSE	DRY
TRUE	TRUE	TRUE	WET

Rules for AND

OR

In everyday life the word 'or' is used in two ways. We can illustrate the inclusive use of **OR** by thinking of a cricket captain looking for players. He might ask "Can you bat or bowl?" He would be pleased if a player could do just one thing well but he would also be pleased if someone could do both. So it is in programming: a compound expression using **OR** is true if either or both of the simple statements or variables are true. Try the following program.

```
100 REMark OR test
110 PRINT "Enter two values" \ "1 for TRUE or 0 for FALSE"
120 INPUT "Can you bat?", batsman
130 INPUT "Can you bowl?", bowler
140 IF batsman OR bowler THEN PRINT "In the team"
```

You can see the effects of different combinations of answers in the rules below:

batsman	bowler	batsman OR bowler	effect
FALSE	FALSE	FALSE	not in team
FALSE	TRUE	TRUE	in the team
TRUE	FALSE	TRUE	in the team
TRUE	TRUE	TRUE	in the team

Rules for OR

When the **inclusive OR** is used a true value in either of the simple statements will produce a true value in the compound expression. If Ian Botham, the England all rounder were to answer the questions both as a bowler and as a batsman, both simple statements would be true and so would the compound expression. He would be in the team.

If you write 0 for false and 1 for true you will get all the possible combinations by counting in binary numbers:

00
01
10
11

NOT

The word **NOT** has the obvious meaning.

NOT true is the same as false
NOT false is the same as true

However you need to be careful. Suppose you hold a red triangle and say that it is:

NOT red **AND** square

In English this may be ambiguous.

If you mean:

(**NOT** red) **AND** square

then for a red triangle the expression is false.

If you mean:

NOT (red **AND** square)

then for a red triangle the whole expression is true. There must be a rule in programming to make it clear what is meant. The rule is that NOT takes precedence over AND so the interpretation:

(**NOT** red) **AND** square

is the correct one This is the same as:

NOT red **AND** square

To get the other interpretation you must use brackets. If you need to use a complex logical expression it is best to use brackets and **NOT** if their usage naturally reflects what you want. But you can if you wish always remove brackets by using the following laws (attributed to Augustus De Morgan)

NOT (a **AND** b) is the same as **NOT** a **OR NOT** b
NOT (a **OR** b) is the same as **NOT** a **AND NOT** b

For example:

NOT (tall **AND** fair) is the same as
NOT tall **OR NOT** fair
NOT (hungry **OR** thirsty) is the same as
NOT hungry **AND NOT** thirsty

Test this by entering

```
100 REMark NOT and brackets
110 PRINT "Enter two values"\"1 for TRUE or 0 for FALSE"
120 INPUT "tall "; tall
130 INPUT "fair "; fair
140 IF NOT (tall AND fair) THEN PRINT "FIRST"
150 IF NOT tall OR NOT fair THEN PRINT "SECOND"
```

Whatever combination of numbers you give as input, the output will always be either two words or none, never one. This will suggest that the two compound logical expressions are equivalent.

XOR-Exclusive OR

Suppose a golf professional wanted an assistant who could either run the shop or give golf lessons. If an applicant turned up with both abilities he might not get the job because the golf professional might fear that such an able assistant would try to take over. He would accept a good golfer who could not run the shop. He would also accept a poor golfer who could run the shop. This is an exclusive OR situation: either is acceptable but not both. The following program would test applicants:

```
100 REMark XOR test
110 PRINT "Enter 1 for yes or 0 for no."
120 INPUT "Can you run a shop?", shop
130 INPUT "Can you teach golf?", golf
140 IF shop XOR golf THEN PRINT "Suitable"
```

The only combinations of answers that will cause the output "Suitable" are (0 and 1) or (1 and 0). The rules for XOR are given below.

Able to run shop	Able to teach	Shop XOR teach	effect
FALSE	FALSE	FALSE	No job
FALSE	TRUE	TRUE	Gets the job
TRUE	FALSE	TRUE	Gets the job
TRUE	TRUE	FALSE	No job

rules for XOR

PRIORITIES

The order of priority for the logical operators is (highest first):

NOT
AND
OR,XOR

For example the expression

rich **OR** tall **AND** fair

means the same as:

rich **OR** (tall **AND** fair)

The **AND** operation is performed first. To prove that the two logical expressions have identical effects run the following program:

```
100 REMark Priorities
110 PRINT "Enter three values"\"Type 1 for Yes and 0 for No"!
120 INPUT rich,tall,fair
130 IF rich OR tall AND fair THEN PRINT "YES"
140 IF rich OR (tall AND fair) THEN PRINT "AYE"
```

Whatever combination of three zeroes or ones you input at line 120 the output will be either nothing or:

```
YES
AYE
```

You can make sure that you test all possibilities by entering data which forms eight three digit binary numbers 000 to 111

```
000 001 010 011 100 101 110 111
```

PROBLEMS ON CHAPTER 10

1. Place ten numbers in a **DATA** statement. **READ** each number and if it is greater than 20 then print it.

2. Test all the numbers from 1 to 100 and print only those which are perfect squares or divisible by 7

3. Toys are described as Safe (S), or Unsafe (U), Expensive (E) or Cheap (C), and either for Girls (G),Boys (B) or Anyone (A). A trio of letters encodes the qualities of each toy. Place five such trios in a DATA statement and then search it printing only those which are safe and suitable for girls.

4. Modify program 3 to print those which are expensive and not safe.

5. Modify program 3 to print those which are safe, not expensive and suitable for anyone.

CHAPTER 11 – HANDLING TEXT – STRINGS

You have used string variables to store character strings and you know that the rules for manipulating string variables or string constants are not the same as those for numeric variables or numeric constants. SuperBASIC offers a full range of facilities for manipulating character strings effectively. In particular the concept of string-slicing both extends and simplifies the business of handling substrings or slices of a string.

ASSIGNING STRINGS

Storage for string variables is allocated as it is required by a program. For example, the lines:

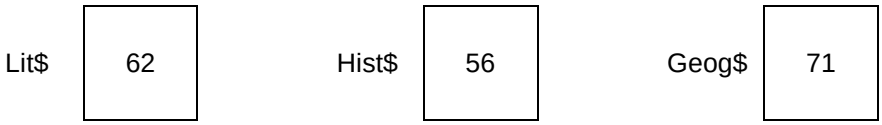
```
100 LET words$ = "LONG"
110 LET words$ = "LONGER"
120 PRINT words$
```

would cause the six letter word, LONGER, to be printed. The first line would cause space for four letters to be allocated but this allocation would be overruled by the second line which requires space for six characters.

It is, however, possible to dimension (i.e. reserve space for) a string variable, in which case the maximum length becomes defined, and the variable behaves as an array.

JOINING STRINGS

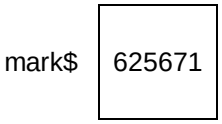
You may wish to construct records in data processing from a number of sources. Suppose, for example, that you are a teacher and you want to store a set of three marks for each student in Literature, History and Geography. The marks are held in variables as shown:



As part of student record keeping you may wish to combine the three string values into one six-character string called *mark$*. You simply write:

```
LET mark$ = lit$ & hist$ & geog$
```

You have created a further variable as shown:

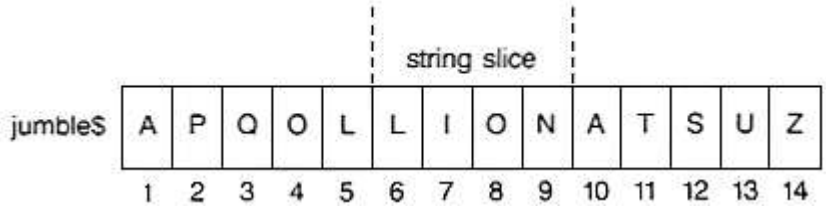


But remember that you are dealing with a character string which happens to contain number characters rather than an actual number. Note that in SuperBASIC the & symbol is used to join strings together whereas in some other BASICs, the + symbol is used for that purpose.

COPY A STRING SLICE

A string slice is part of a string. It may be anything from a single character to the whole string. In order to identify the string slice you need to know the positions of the required characters.

Suppose you are constructing a children's game in which they have to recognise a word hidden in a jumble of letters. Each letter has an internal number - an index - corresponding to its position in the string. Suppose the whole string is stored in the variable jumble$ and the clue is Big cat.



You can see that the answer is defined by the numbers 6 to 9 which indicate where it is. You can abstract the answer as shown :

```
100 jumble$ = "APQOLLIONATSUZ"
110 LET an$ = jumble$(6 TO 9)
120 PRINT an$
```

REPLACE A STRING SLICE

Now suppose that you wish to change the hidden animal into a bull. You can write two extra lines:

```
130 LET jumble$(6 TO 9) = "BULL"
140 PRINT jumble$
```

The output from the whole five-line program is:

```
LION
APQOLBULLATSUZ
```

All string variables are initially empty, they have length zero. If you attempt to copy a string into a string-slice which has insufficient length then the assignment may not be recognised by SuperBASIC.

If you wish to copy a string into a string-slice then it is best to ensure the destination string is long enough by padding it first with spaces.

```
100 LET subject$ = "ENGLISH MATHS COMPUTING"
110 LET student$ = ""
120 LET student$(9 TO 13) = subject$(9 TO 13)
```

We say that "BULL" is a slice of the string "APQOLBULLATSUZ". The defining phrase:

```
(6 TO 9)
```

is called a **slicer**. It has other uses. Notice how the same notation may be used on both sides of the **LET** statement. If you want to refer to a single character it would be clumsy to write:

```
jumble$(6 TO 6)
```

just to pick out the "B" (possibly as a clue) so you can write instead:

```
jumble$(6)
```

to refer to a single character

COERCION

Suppose you have a variable, mark$ holding a record of examination marks. The slice giving the history mark may be extracted and scaled up, perhaps because the history teacher has been too strict in the marking. The following lines will extract the history mark:

```
100 LET mark$ = "625671"
110 LET hist$ = mark$(3 TO 4)
```

The problem now is that the value "56" of the variable, *hist$* is a string of characters not numeric data. If you want to scale it up by multiplying by say 1.125, the value of *hist$* must be converted to numeric data first, SuperBASIC will do this conversion automatically when we type:

```
120 LET num = 1 .125 * hist$
```

Line 120 converts the string "56" to the number 56 and multiplies it by 1.125 giving 63.

Now we should replace the old mark by the new mark but now the new mark is still the number 63 and before it can be inserted back into the original string it must be converted back to the string '63'. Again SuperBASIC will convert the number automatically when we type:

```
130 LET mark$(3 TO 4) = num
140 PRINT mark$
```

The output from the whole program is:

```
626371
```

which shows the history mark increased to 63.

Strictly speaking it is illegal to mix data types in a **LET** statement. It would be silly to write:

```
LET num = "LION"
```

and you would get an error message if you tried, but if you write:

```
LET num = "65"
```

the system will conclude that you want the number 65 to become the value of num and do that. The complete program is:

```
100 LET mark$ = "625671"
110 LET hist$ = mark$(3 TO 4)
120 LET num = 1.125 * hist$
130 LET mark$(3 TO 4) = num
140 PRINT mark$
```

Again the output is the same!

In line 120 a string value was converted into numeric form so that it could be multiplied; In line 130 a number was converted into string form. This converting of data types is known as **type coercion**.

You can write the program more economically if you understand both string-slicing and coercion now:

```
100 LET mark$ = "625671"
110 LET mark$(3 TO 4) = 1 .125 * mark$(3 TO 4)
120 PRINT mark$
```

If you have worked with other BASICs you will appreciate the simplicity and power of string-slicing and coercion.

SEARCHING A STRING

You can search a string for a given substring. The following program displays a jumble of letters and invites you to spot the animal.

```
100 REM Animal Spotting
110 LET jumble$ = "SYNDICATE"
120 PRINT jumble$
```

```
130 INPUT "What is the animal?" ! an$
140 IF an$ INSTR jumble$ AND an$(1) = "C"
150   PRINT "Correct"
160 ELSE
170   PRINT "Not correct"
180 END IF
```

The operator INSTR, returns zero if the guess is incorrect. If the guess is correct INSTR returns the number which is the starting position of the string-slice, in this case 6.

Because the expression:

```
an$ INSTR jumble$
```

can be treated as a logical expression the position of the string in a successful search can be regarded as true, while in an unsuccessful search it can be regarded as **false**.

OTHER STRING FUNCTIONS

You have already met **LEN** which returns the length (number of characters) of a string. You may wish to repeat a particular string or character several times. For example, if you wish to output a row of asterisks, rather than actually enter forty asterisks in a PRINT statement or organise a loop you can simply write:

```
PRINT FILL$ ("*",40)
```

Finally it is possible to use the function CHR$ to convert internal codes into string characters. For example:

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

would output A.

COMPARING STRINGS

A great deal of computing is concerned with organising data so that it can be searched quickly. Sometimes it is necessary to sort it in to alphabetical order. The basis of various sorting processes is the facility for comparing two strings to see which comes first. Because the letters A,B,C ... are internally oded as 65,66,67 .... it is natural to regard as correct the following statements:

A is less than B
B is less than C

and because internal character by character comparison is automatically provided:

CAT is less than DOG
CAN is less than CAT

You can write, for example:

```
IF "CAT" < "DOG" THEN PRINT "MEOW"
```

and the output would be:

```
MEOW
```

Similarly:

```
IF "DOG" > "CAT" THEN PRINT "WOOF"
```

would give the output:

```
WOOF
```

We use the comparison symbols of mathematics for string comparisons. All the following logical statements expressions are both permissible and true.

```
"ALF" < "BEN"
"KIT" > "BEN"
"KIT" <= "LEN"
"KIT" >= "KIT"
"PAT" >= "LEN"
"LEN" <= "LEN"
"PAT" <> "PET"
```

So far comparisons based simply on internal codes make sense, but data is not always conveniently restricted to upper case letters. We would like, for example:

Cat to be less than COT
and K2N to be less than K27N

A simple character by character comparison based on internal codes would not give these results, so SuperBASIC behaves in a more intelligent way. The following program, with suggested input and the output that will result, illustrates the rules for comparison of strings.

```
100 REMark comparisons
110 REPeat comp
120   INPUT "input a string" ! first$
130   INPUT "input another string" ! second$
140   IF first$ < second$ THEN PRINT "Less"
150   IF first$ > second$ THEN PRINT "Greater"
160   IF first$ = second$ THEN PRINT "Equal"
170 END REPeat comp
```

Input		Output
CAT	COT	Greater
CAT	CAT	Equal
PET	PETE	Less
K6	K7	Less
K66	K7	Greater
K12N	K6N	Greater

> Greater than - Case dependent comparision, numbers compared in numerical order

< Less than - Case dependent, numbers compared in numerical order

= Equals - Case dependent, strings must be the same

== Equivalent - String must be 'almost' the same, Case independent, numbers compared in numerical order

>= Greater than or equal to - Case dependent, numbers compared in numerical order

<= Less than or equal to Case dependent, numbers compared in numerical order.

PROBLEMS ON CHAPTER 11

1. Place 12 letters, all different, in a string variable and another six letters in a second string variable. Search the first string for each of the six letters in turn saying in each case whether it is found or not found.

2. Repeat using single character arrays instead of strings. Place twenty random upper case letters in a string and list those which are repeated.

3. Write a program to read a sample of text all in upper case letters. Count the frequency of each letter and print the results.

   "GOVERNMENT IS A TRUST, AND THE OFFICERS OF THE GOVERNMENT ARE TRUSTEES; AND BOTH THE TRUST AND THE TRUSTEES ARE CREATED FOR THE BENEFIT OF THE PEOPLE. HENRY CLAY 1829."

4. Write a program to count the number of words in the following text. A word is recognised because it starts with a letter and is followed by a space, full stop or other punctuation character.

   "THE REPORTS OF MY DEATH ARE GREATLY EXAGGERATED. CABLE FROM MARK TWAIN TO THE ASSOCIATED PRESS, LONDON 1896."

5. Rewrite the last program illustrating the use of logical variables and procedures.

CHAPTER 12 – SCREEN OUTPUT

SuperBASIC has so extended the scope and variety of facilities for screen presentation that we describe the features in two sections: *Simple Printing* and *Screen*.

The first section describes the output of ordinary text. Here we explain the minimal well established methods of displaying messages, text, or numerical output. Even in this mundane section there is innovation in the concept of the 'intelligent' space an example of combining ease of use with very useful effects.

The second section is much bigger because it has a great deal to say. The wide range of features actually makes things easier For example, you can draw a circle by simply writing the word **CIRCLE** followed by a few details to define such things as its position and size. Many other systems require you to understand some geometry and trigonometry in order to do what is, in concept, simple.

Each keyword has been carefully chosen to reflect the effect it causes. **WINDOW** defines an area of the screen: **BORDER** puts a border round it; **PAPER** defines the background colour; **INK** determines the colour of what you put on the paper.

If you work through this chapter and get a little practice you will easily remember which keyword causes which effect. You will add that extra quality to your programming fairly easily. With experience you may see why computer graphics is becoming a new art form.

SIMPLE PRINTING

The keyword **PRINT** can be followed by a sequence of print items. A print item may be any of:
text such as: "This is text"

variables such as : num, word$
expressions such as : 3 * num, day$ & week$

Print items may be mixed in any print statement but there must be one or more print separators between each pair. Print separators may be any of:

; No effect - it just separates print items.

! Normally inserts a space between output items. If an item will not fit on the current line it behaves as a new line symbol. If the item is at the start of line a space is not generated.

, A tabulator causes the output to be tabulated in columns of 8 characters

\ A new line symbol will force a new line.

TO Allows tabbing.

The numbers 1,2,3 are legitimate print items and are convenient for illustrating the effects of print separators

Statement	Effect
`100 PRINT 1,2,3`	`1 2 3`
`100 print 1 ! 2 ! 3 !`	`1 2 3`

`100 PRINT 1 \ 2 \ 3`	`1`
	`2`
	`3`

```
100 PRINT 1 ; 2 ; 3          123

100 PRINT "This is text"     This is text

100 LET word$ = " "          Moves print position
110 PRINT word$

100 LET num = 13             13
110 PRINT num

100 LET an$ = "yes"          I say yes
110 PRINT "I say" ! an$

110 PRINT"Sum is" ! 4+2      Sum is 6
```

You can position print output anywhere on the screen with the **AT** command.

For example:

```
AT 10,15 : PRINT "This is on row 10 at column 15"
```

The **CURSOR** command can be used to position the print output anywhere on the screen's scale system. For example:

```
CURSOR 100,150 : PRINT "this is 100 pixel grid units across and 150
down"
```

If you read the *Keyword Reference Guide* you may find it difficult to reconcile the section on **PRINT** with the above description. Two of the difficulties disappear if you understand that:

Text in quotes, variables and numbers are all strictly speaking, expressions: they are the simplest (degenerate) forms of expressions.

Print separators are strictly classified as print items.

SCREEN

This section introduces general effects which apply whether you wish to output text or graphics. The statement:

```
MODE 8 or MODE 256
```

will select **MODE 8** in which there are:

256 pixels across numbered 0 511 (two numbers per pixel)
256 pixels down numbered 0-255
8 colours

A pixel is the smallest area of colour which can be displayed. We use the term, **solid colour** because these start with ordinary solid-looking colours of which there are only eight. However, by using various effects a variety of shades and textures can be achieved. If you are using your QL with an ordinary television set then the television set will not be able to reproduce any of these extra effects.

The statement:

```
MODE 4 or MODE 512
```

will select **MODE 4** in which there are:

512 pixels across numbered 0 to 511
256 pixels down numbered 0 to 255
4 colours

COLOUR

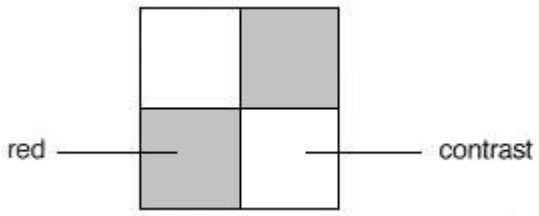
You can select a colour by using the following code in combination with suitable keywords such as **PAPER**, **INK** etc. Note that the numbers by themselves mean nothing. The numbers are only interpreted as colours when they are used with **PAPER** and **INK**, etc.

8 Colour Mode	Code	4 Colour Mode
Black	0	Black
Blue	1	Black
Red	2	Red
Magenta	3	Red
Green	4	Green
Cyan	5	Green
Yellow	6	White
white	7	white

For example **INK 3** would give magenta in **MODE 8**.

STIPPLES

You can if you wish specify two colours in a suitable statement. For example 2,4 would give a chequerboard stipple as shown. In each group of four pixels two would be red (code 2) corresponding to the colour selected first. The other two pixels would be a contrast It is not really possible to display this effect on a domestic television set.



If you write:

```
INK 2,4
```

the mix colour is formed from the two codes 2 and 4. We will call these choices colour and contrast!

```
INK colour, contrast
```

You can find out what the stipple effects are by trying them but we give more technical details below.

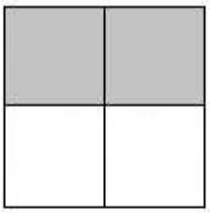
```
100 REMark Colour/Contrast
110 FOR colour = 0 TO 7 STEP 2
120   PAPER colour : CLS
140   FOR contrast = 0 TO 7 STEP 2
150     BLOCK 100,50,40,50,colour,contrast
160     PAUSE 50
170   END FOR contrast
```

```
180 END FOR colour
```

If you wish to try different stipples you can add a third code number to the colour specification. For example:

```
INK 2,4,1
```

would specify a red and green horizontal stripe effect. A block of four pixels would be:



The possible effects are shown using red and contrast

Code	Name	Effect
0	Single pixel of contrast	
1	Horizontal Stripes	
2	Vertical Stripes	
3	Chequerboard	

Stipple Patterns

COLOUR PARAMETERS

You can specify a colour/stipple effect as described above by using three numbers. For example:

```
INK colour, contrast, stipple
```

could be used with :

colour in range 0 to 7
contrast in range 0 to 7
stipple in range 0 to 3

You could achieve the same effect with a single number if you wish though it is not so easy to construct. See the *Concept Reference Guide - colour*.

The following program will display all the possible colour effects:

```
100 REMark Colour Effects
110 FOR num = 0 TO 255
120   BLOCK 100,50,40,50,num
130   PAUSE 50
140 END FOR num
```

PAPER

PAPER followed by one, two or three numbers specifies the background. For example:

```
PAPER 2       {red}
PAPER 2,4     {red/green chequerboard}
PAPER 2,4,1   {red/green horizontal stripes}
```

The colour will not be visible until something else is done, for example, the screen is cleared by typing **CLS**.

INK

INK followed by one, two or three numbers specifies the colour for printing characters, lines or other graphics. The colour and stipple effects are the same as for **PAPER**. For example:

```
INK 2         {red ink}
INK 2,4       {red/green chequerboard ink 3}
INK 2,4,1     {red/green horizontal striped ink}
```

The ink will be changed for all subsequent output.

CLS

CLS means clear the window to the current paper colour - like a teacher cleaning a blackboard, except that it is electronic and multi-coloured.

FLASHING

You can make the ink colour flash in mode 8 only. To turn flash on you might type:

```
FLASH 1
```

and to turn it off:

```
FLASH 0
```

Allowing flashing characters to overlap can produce alarming results.

FILES

You will have used Microdrives for storing programs and you will have used the commands **LOAD** and **SAVE**. Cartridges can be used for storing data as well as programs. The word file usually means a sequence of data records, a record being some set of related information such as name, address and telephone number.

Two of the most widely used types of file are serial and direct access files. Items in a serial file are usually read in sequence starting with the first. If you want the fiftieth record you have to read the first forty-nine in order to find it. On the other hand the fiftieth record in a direct access file can be found quickly because the system does not need to work through the earlier records to get it. Pop music on a cassette is like a serial file but eight pieces on a long playing record form a direct access file. You can move the pick up arm directly onto any of the eight tracks.

The simplest possible type of file is just a sequence of numbers. To illustrate the idea we will place the numbers 1 to 100 in a file called numbers. However the complete file name is made up of two parts:

```
device name
appended information
```

Suppose that we wish to create the file, *numbers*, on a cartridge in Microdrive 1. The device name is:

```
mdv1_
```

and the appended information is just the name of the file:

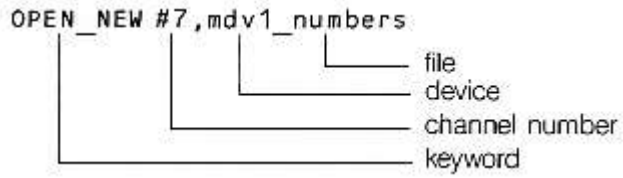
```
numbers
```

So the complete file name is:

```
mdv1_numbers
```

CHANNELS

It is possible for a program to use several files at once, but it is more convenient to refer to a file by an associated channel number This can be any integer in the range 0 to 15. A file is associated with a channel number by using the **OPEN** statement or, if it is a new file, **OPEN_NEW**. For example you may choose channel 7 for the numbers file and write:



You can now refer to the file just by quoting the number #7. The complete program is:

```
100 REMark simple file
110 OPEN_NEW #7,mdv1_numbers
120 FOR number = 1 TO 100
130   PRINT #7,number
140 END FOR number
150 CLOSE #7
```

The PRINT statement causes the numbers to be 'printed' on the cartridge file because #7 has been associated with it. The CLOSE #7 statement is necessary because the system has some internal work to do when the file has been used. It also releases channel 7 for other possible uses. After the program has executed type

```
DIR mdv1_
```

and the directory should show that the file numbers exists on the cartridge in Microdrive mdv1_ .

You also need to know that the file is correct and you can only be certain of this if the file is read and checked. The necessary keyword is OPEN_IN, otherwise the program for reading data from a file is similar to the previous one.

```
100 REMark Reading a file
110 OPEN IN #6, mdv1_numbers
120 FOR item = 1 TO 100
130 INPUT #6, number
140 PRINT ! number !
150 END FOR item
160 CLOSE #6
```

The program should output the numbers 1 to 100, but only if the cartridge containing the file "numbers" is still in Microdrive mdv1_.

DEVICES AND CHANNELS

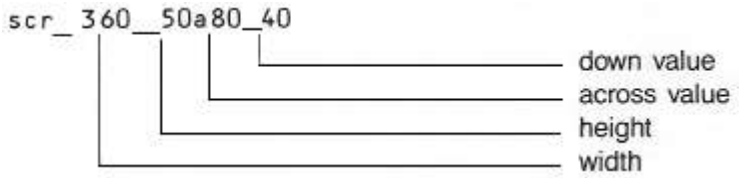
You have seen one example of a device, a file of data on a Microdrive. We may say loosely that a file has been opened but strictly we mean that a device has been associated with a particular channel. Any further necessary information has also been provided. Certain devices have channels permanently associated with them by the system:

Channel	Use
#0	OUTPUT – command window
#1	INPUT – keyboard
#2	OUTPUT – print window LIST – list output

You can create a window of any size anywhere on the screen. The device name for a window is:

```
scr
```

and the appended information is, for example:



The following program creates a window with the channel number 5 and fills it with green (code 4) and then closes it:

```
100 REMark Create a window
110 OPEN #5, scr_400x200a20x50
120 PAPER #5,4 : CLS #5
130 CLOSE #5
```

Notice that each window can have its own features such as paper ink, etc. The fact that a window has been opened does not mean that it is the current default window.

You can change the position or shape of an opened window without closing it and reopening it. Try adding two lines to the previous program:

```
124 WINDOW #5,300,100,110,65
126 PAPER #5,2 : CLS #5
```

Re-run the program and you will find a red window within the original green one. This red window is now the one associated with channel 5, see figure.

BORDER

You can place a border round the edge of the screen or a window. For example:

```
BORDER #5,6
```

would create a border round the channel #5 window. It would be 6 units thick and the size of the window would be correspondingly reduced. The border would be transparent, protecting anything that was under it. You can specify a coloured border by the usual method.

```
BORDER #5,6,2
```

would produce a red border. You can make a border of other colours and textures by the usual methods. For example,

```
BORDER 10
```

Will add a 10 pixel thick transparent border to the current window (transparent because no colour was specified) and

```
BORDER 2,0,7,0
```

Will add a 2 pixel thick black and white stipple border.

BLOCK

You can specify a block's size, position and colour with a single statement. It is placed in the pixel co-ordinate system relative to the current window or screen. For example:

```
BLOCK #5,10,20,50,100,2
```

would create a block in the # 5 window at a position 50 units across and 100 units down. It would be 10 units wide and 20 units high. Its colour would be red.

It is worth noting that **WINDOW** and **BLOCK** statements work without alteration in 4 and 8 colour mode (though the colours may vary) because the across values are always on a 0 to 511 scale and there are always 256 pixel positions down.

SPECIAL PRINTING CSIZE

You can alter the size of characters. For example:

```
CSIZE 3,1
```

will give the largest possible characters and:

```
CSIZE 0,0
```

will give the smallest. The first number must be 0, 1, 2 or 3 and determines the width. The second must be 0 or 1 and determines the height. The normal sizes are:

```
MODE 4    CSIZE 0,0

MODE 8    CSIZE 2,0
```

The number of lines and columns available for each character size is dependent on whether the output is viewed on a monitor or on a television set: the row and column sizes given are for a monitor; those for a television set will be smaller and also will vary between different televisions.

If you are using low resolution mode the QL will not allow you to select a character size smaller than default size.

STRIP

You can provide a special background for characters to make them stand out. For example:

```
STRIP 7
```

will give a white strip while

```
STRIP 2,4,2
```

will give a red/green vertical striped strip. All the normal colour combinations are possible.

OVER

Normally printing occurs on the current paper colour. You can alter this by using strip. You can make further effects by using:

```
OVER 1      1 prints in ink on a transparent strip
OVER -1     -1 prints in ink over existing display on screen
```

To revert to normal printing on current strip use:

```
OVER 0
```

UNDER

You can underline characters.

```
UNDER 1     underlines all subsequent output in the current ink
UNDER 0     switches off underling.
```

SCALE GRAPHICS

If you wish to draw reasonably true geometric figures on a TV or video screen you cannot easily use a pixel-based system. If you use **scale graphics** then the system will do the necessary work to ensure that you can fairly easily draw reasonable circles, squares and other shapes.

The default scale of the graphics coordinate system is 100 in the vertical direction and whatever is needed in the across direction to ensure that shapes drawn with the special graphics keywords (**PLOT, DRAW, CIRCLE**) are true.

The **graphics origin** is not the same as the pixel origin which is used to define the position of windows and blocks. The graphics origin is at the bottom left hand corner of the current screen or window.

POINTS AND LINES

It is easy to draw points and lines using scale graphics. Using a vertical scale of 100 a point near the centre of the window can be plotted with:

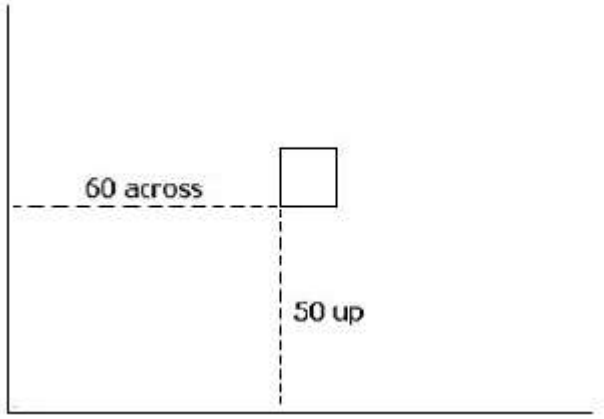
```
POINT 60,50
```

The point (60 units across and 50 units up) will be plotted in the current ink colour. Similarly a line may be drawn with the statement:

```
LINE 60,50 TO 80,90
```

Further elements can be added. For example, the following will draw a square:

```
LINE 60,50 TO 70,50 TO 70,60 TO 60,60 TO 60,50
```



RELATIVE MODE

Pair of coordinates such as:

> across, up

normally define a point relative to the origin 0,0 in the bottom left hand corner of a window (or elsewhere if you choose). It is sometimes more convenient to define points relative to the current cursor position. For example the square above may be plotted in another way using the **LINE_R** statement which means:

> "Make all pairs of coordinates relative to the current cursor position."

```
POINT 60,50
LINE_R 0,0 TO 10,0 TO 0,10 TO -10,0 TO 0,-10
```

First the point 60,50 becomes the origin, then, as lines are drawn, the end of a line becomes the origin for the next one.

The following program will plot a pattern of randomly placed coloured squares.

```
100 REMark Coloured Squares
110 PAPER 7 : CLS
120 FOR sq = 1 TO 100
130   INK RND(1 TO 6)
140   POINT RND(90),RND(90)
150   LINE R 0,0 TO 10,0 TO 0,10 TO -10,0 TO 0,-10
160 END FOR sq
```

The same result could be achieved entirely with absolute graphics but it would require a little more effort.

CIRCLES AND ELLIPSES

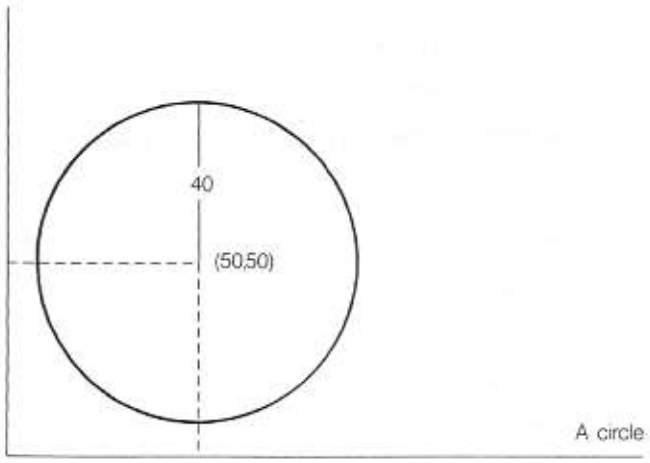
If you want to draw a circle you need to specify:

> position say 50,50
> radius say 40

The statement

```
CIRCLE 50,50,40
```

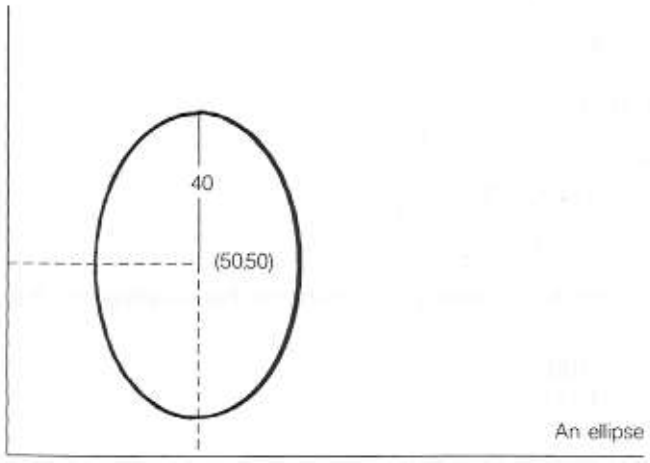
will draw a circle with the centre at position 50,50 and radius (or height) 40 units, see figure:



If you add two more parameters:

e.g. `CIRCLE 50,50,40,.5`

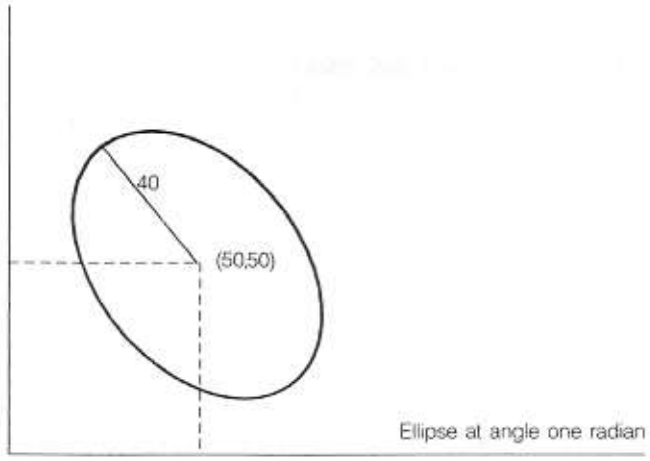
You will get an ellipse. The keywords **CIRCLE** and **ELLIPSE** are interchangeable.



The height of the ellipse is 40 as before but the horizontal 'radius' is now only 0.5 of the height. The number 0.5 is called the eccentricity. If the eccentricity is 1 you get a circle if it is less than 1 and greater than zero you get an ellipse. If you want to tilt an ellipse you can change the fifth parameter, for example:

```
CIRCLE 50,50,40,.5,1
```

This will tilt the ellipse anti-clockwise by one radian, about 57 degrees, as shown in figure below



A straight angle is 180 degrees or PI radians, so you can make a pattern of ellipses with the program:

```
100 FOR rot = 0 TO 2*PI STEP PI/6
110   CIRCLE 50,50,40,0.5,rot
120 END FOR rot
```

The order of the parameters for a circle or ellipse is:

centre _across, centre_up, height [eccentricity, angle]

The last two parameters are optional and this is indicated by putting them inside square brackets ([ ]). Write a program which does the following:

1. Open a window 100x100 at (100,50)
2. Scale 100 in mode 8
3. Select black paper and clear window
4. Make green border 2 units wide
5. Draw a pattern of six coloured circles.
6. Close the window

```
100 REMark pattern
110 MODE 8
120 OPEN #7,scr_100x100a100x50
130 SCALE #7,100,0,0
140 PAPER #7,0 : CLS #7
150 BORDER #7,2,4
160 FOR colour = 1 TO 6
170   INK #7,colour
180   LET rot = 2*PI/colour
190   CIRCLE #7,50,50,30,0.5,rot
200 END FOR colour
210 CLOSE #7
```

You can get some interesting effects by altering the program. For example try the amendments:

```
160 FOR colour = 1 TO 100
180 LET rot = colour*PI/50
```

ARCS

If you want to draw an arc you need to decide:

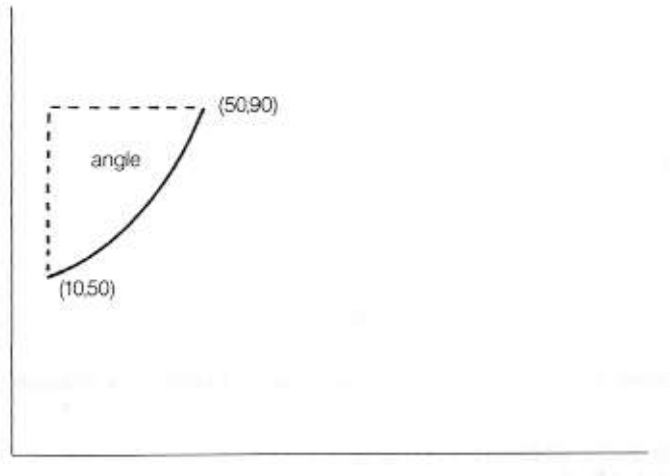
starting point
end point

amount of curvature

The first two items are straightforward but the amount of curvature is not so easy. You can do it by drawing accurately or by trial and error but you must decide what angle the arc subtends and then specify the angle in radians. An angle of 1.5 radians would give a sharp bend and a small angle would give a very gentle curvature. Try for example:

```
ARC 10,50 TO 50,90,1
```

which gives a moderate curvature in the current **INK** colour.



FILL

You can fill a closed shape with the current **INK** colour by simply writing:

```
FILL 1
```

before the shape is drawn. The following program produces a green circle.

```
INK 4
FILL 1
CIRCLE 50,50,30
```

The **FILL** command works by drawing touching horizontal lines between suitable points. The statement:

```
FILL 0
```

Will turn off the **FILL** effect.

SCROLLING AND PANNING

You can scroll or pan the display in a window like a film cameraman. You arrange scrolling In terms of pixels. A positive number of pixels indicates upwards scrolling, thus

```
SCROLL 10
```

Moves the display in the current window or screen 10 pixels downwards.

```
SCROLL -8
```

Moves the display 8 pixels up. You can add a second parameter to induce part-scrolling.

```
SCROLL -8, 1
```

Will scroll the part above (not including) the cursor line and:

```
SCROLL -8, 2
```

Will scroll the part below (not including) the cursor line.

As scrolling occurs, the space left by movement of the display is filled with the current Paper colour. A second parameter 0 has no effect.

You can **PAN** the display in the current window left or right. The **PAN** statement works In a similar manner to scroll but

```
Pan 40    moves display right
Pan -40   moves display left
```

A second parameter gives a partial **PAN**

0 - whole screen
3 - the whole of the line occupied by the cursor
4 - the right hand side of the line occupied by the cursor. The area of the cursor is also included.

If you are using stipples or are in 8 colour mode then windows must be panned or Scrolled in multiples of 2 pixels.

PROBLEMS ON CHAPTER 12

1. Write a program which draws a 'Snakes and Ladders' grid of ten rows of ten rows of ten squares.

2. Place the numbers 1 to 100 in the squares starting at the bottom left and place F for finish in the last square.

3. Draw a dartboard on the screen. It should consist of an outer ring which could hold numbers. A 'doubles' ring and 'triples' ring as shown and a centre consisting of a 'bull's eye' and a ring around it.

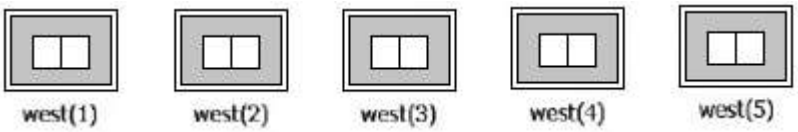
CHAPTER 13 – ARRAYS

Suppose you are a prison governor and you have a new prison block which is called the West Block. It is ready to receive 50 new prisoners. You need to know which prisoner (known by his number) is in which cell. You could give each cell a name but it is simpler to give them numbers 1 to 50.

In a computing simulation we will imagine just 5 prisoners with numbers which we can put in a **DATA** statement:

```
Data 50, 37, 86, 41, 32
```

We set up an array of variables which share the name, west, and are distinguished by a number appended in brackets.



It is necessary to declare an array and give its dimensions with a **DIM** statement:

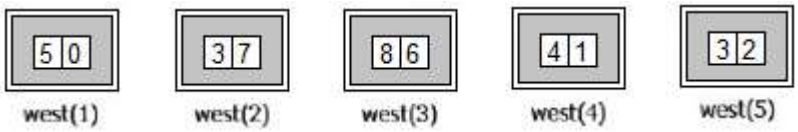
```
DIM west(5)
```

This enables SuperBASIC to allocate space, which might be a large amount. After the **DIM** statement has been executed the five variables can be used.

The convicts can be **READ** from the **DATA** statement into the five array variables:

```
FOR cell = 1 TO 5 : READ west (cell)
```

We can add another **FOR** loop with a **PRINT** statement to prove that the convicts are in the cells.



The complete program is shown below:

```
100 REMark Prisoners
110 DIM west(5)
120 FOR cell 1 = 1 TO 5 : READ west(cell)
130 FOR cell = 1 TO 5 : PRINT cell ! west(cell)
140 DATA 50, 37, 86, 41, 32
```

The output from the program is:

```
1 50
2 37
3 86
4 41
5 32
```

The numbers 1 to 5 are called *subscripts* if the array name, *west.* The array *west,* is a numeric array consisting of five numeric array elements.

You can replace line 130 by:

```
130 PRINT west
```

This will output the values only:

```
0
50
37
86
41
32
```

The zero at the top of the list appears because subscripts range from zero to the declared number. We will show later how useful the zero elements in arrays can be. Note also that when a numeric array is DIMensioned its elements are all given the value zero.

STRING ARRAYS

String arrays are similar to numeric arrays but an extra dimension in the **DIM** statement specifies the length of each string variable in the array. Suppose that ten of the top players at Royal Birkdale for the 1982 British Golf Championship were denoted by their first names and placed in **DATA** statements.

```
DATA "Tom","Graham","Sevvy","Jack","Lee"
DATA "Nick","Bernard","Ben","Gregg","Hal"
```

You would need ten different variable names, but if there were a hundred or a thousand players the job would become impossibly tedious. An array is a set of variables designed to cope with problems of this kind. Each variable name consists of two parts:

a name according to the usual rules
a numeric part called a subscript

Write the variable names as:

```
flat$(1),flat$(2),flat$(3)...etc
```

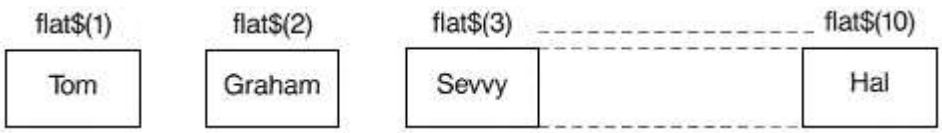
Before you can use the array variables you must tell the system about the array and its dimensions:

```
DIM flat$(10,8)
```

This causes eleven (0 to 10) variables to be reserved for use in the program. Each string variable in the array may have up to eight characters. **DIM** statements should usually be placed all together near the beginning of the program. Once the array has been declared in a **DIM** statement all the elements of the array can be used. One important advantage is that you can give the numeric part (the subscript) as a numeric variable. You can write:

```
FOR number = 1 TO 10 : READ flat$(number)
```

This would place the golfers in their 'flats':



You can refer to the variables in the usual way but remember to use the right subscript. Suppose that Tom and Sevvy wished to exchange flats. In computing terms one of them, Tom say, would have to move into a temporary flat to allow Sevvy time to move. You can write:

```
LET temp$ = flat$(1) : REMark Tom into temporary
LET flat$(1) = flat$(3) : REMark Sevvy into flat$(1)
LET flat$(3) = temp$ : REMark Tom into flat$(3)
```

The following program places the ten golfers in an array named flat$ and prints the names of the occupants with their 'flat numbers' (array subscripts) to prove that they are in residence. The occupants of flats 1 and 3 then change places. The list of occupants is then printed again to show that the exchange has occurred.

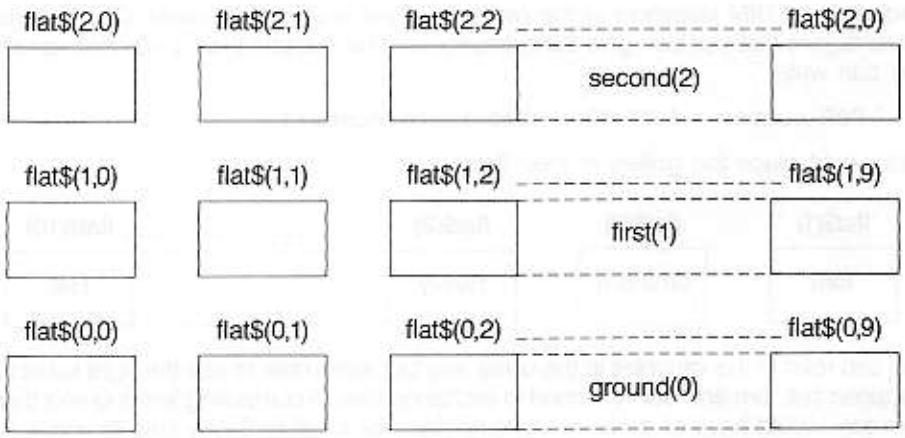
```
100 REMark Golfers' Flats
110 DIM flat$(10,8)
120 FOR number = 1 TO 10 : READ flat$(number)
130 printlist
140 exchange
150 printlist
160 REMark End of main program
170 DEFine PROCedure printlist
180 FOR num = 1 TO 10 : PRINT num,flat$(num)
190 END DEFine
200 DEFine PROCedure exchange
210 LET temp$ = f1at$(1)
220 LET flat$(1) = f1at$(3)
230 LET flat$(3) = temp$
240 END DEFine
250 DATA "Tom","Graham","Sevvy","Jack","Lee"
260 DATA "Nick","Bernard","Ben","Greg","Hal"
```

output (line 130)	output (line 150)
1 Tom	1 Sevvy
2 Graham	2 Graham
3 Sevvy	3 Tom
4 Jack	4 Jack
5 Lee	5 Lee
6 Nick	6 Nick
7 Bernard	7 Bernard
8 Ben	8 Ben
9 Gregg	9 Gregg
10 Hal	10 Hal

TWO DIMENSIONAL ARRAYS

Sometimes the nature of a problem suggests two dimensions such as 3 floors of 10 flats rather than just a single row of 30.

Suppose that 20 or more golfers need flats and there is a block of 30 flats divided into three floors of ten f lats each. A realistic method of representing the block would be with a two-dimensional array, You can think of the thirty variables as shown below:



Assuming **DATA** statements with 30 names, a suitable way to place the names in the flats is:

```
120 FOR floor = 0 TO 2
130   FOR num = 0 TO 9
140     READ flats$(floor,num)
150   END FOR num
160 END FOR floor
```

You also need a DIM statement:

```
20 DIM flat$(2,9,8)
```

which shows that the first subscript can be from 0 to 2 (floor number) and the second subscript can be from 0 to 9 (room number). The third number states the maximum number of characters in each array element.

We add a print routine to show that the golfers are in the flats and we use letters to save space.

```
100 REMark 30 Golfers
110 DIM flat$(2,9,8)
120 FOR floor = 0 TO 2
130   FOR num = 0 TO 9
140     READ flat$(floor,num) : REMark Golfer goes in
150   END FOR num
160 END FOR floor
170 REMark End of input
180 FOR floor = 0 TO 2
190   PRINT "Floor number" ! Floor
200   FOR num = 0 TO 9
210     PRINT 'Flat' ! num ! flat$(floor,num)
220   END FOR num
230 END FOR floor
240 DATA "A","B","C","D","E","F","G","H","I","J"
250 DATA "K","L","M","N","O","P","Q","R","S","T"
260 DATA "U","V","W","X","Y","Z","@","£","$","%"
```

The output starts:

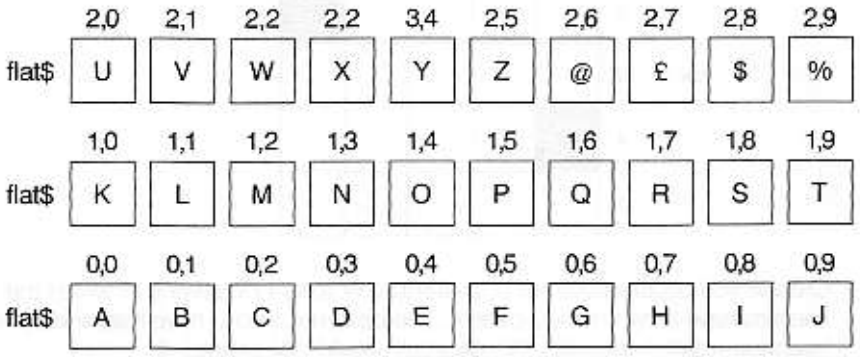
```
Floor number 0
Flat 0 A
Flat 1 B
Flat 2 C
```

And continues giving the thirty occupants.

ARRAY SLICING

You may find this section hard to read though it is essentially the same concept as string slicing. You will probably need string-slicing if you get beyond the learning stage of programming. The need for array-slicing is much rarer and you may wish to omit this section particularly on a first reading.

We now use the golfers' flats to illustrate the concept of array slicing. The flats will be numbered 0 to 9 to keep to single digits and names will be single characters for space reasons.



Given the above values the following are array slices:

flat$(1,3)	Means a single array element with value N
flat$(1,1 TO 6)	Means six elements with values L M N 0 P Q

Array Element	Value
`flat$(1,1)`	L
`flat$(1,2)`	M
`flat$(1,3)`	N
`flat$(1,4)`	O
`flat$(1,5)`	P
`flat$(1,6)`	Q

`flat$(1)`	means flat$(1,0 TO 9) Ten elements with values K L M N O P Q R S T

In these examples a range of values of a subscript can be given instead of a single value. If a subscript is missing completely the complete range is assumed. In the third example the second subscript is missing and it is assumed by the system to be **0 TO 9**.

The techniques of array slicing and string slicing are similar though the latter is more widely applicable.

PROBLEMS ON CHAPTER 13

1. SORTING

Place ten numbers in an array by reading from a DATA statement. Search the array to find the lowest number. Make this lowest number the value of the first element of a new array. Replace it in the first array with a very large number. Repeat this process making the second lowest number the second value in the new array and so on until you have a sorted array of numbers which should then be printed.

2. SNAKES AND LADDERS

Represent a snakes and ladders game with a 100 element numeric array. Each element should contain either

zero

or:

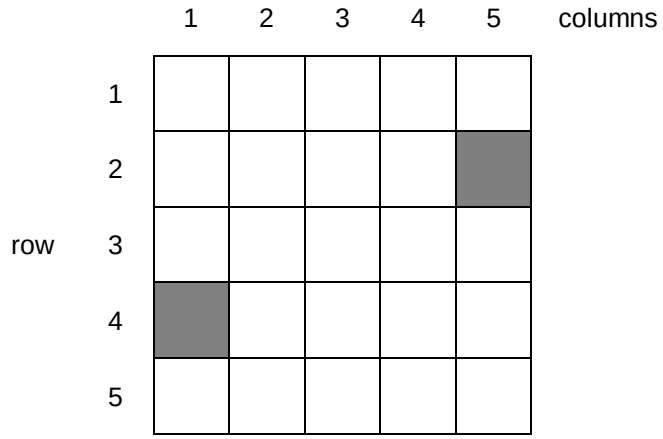
a number in the range 10 to 90 meaning that a player should transfer to that number by going 'up a ladder' or 'down a snake'

or:

the digits 1, 2, 3, etc. to denote a particular player's position.

Set up six snakes and six ladders by placing numbers in the array and simulate one 'solo' run by a single player to test the game.

3. CROSSWORD BLANKS



Crosswords usually have an odd number of rows or columns in which the black squares have a symmetrical pattern. The pattern is said to have rotational symmetry because rotation through 180 degrees would not change it.
Note that after rotation through 180 degrees the square in row 4, column1 could become the square in row 2, column 5. That is row 4, column 1 becomes row 2, column 5 in a 5 x 5 grid.

Write a program to generate and display a symmetrical pattern of this kind.

4. Modify the crossword pattern so that there are no sequences, across or down, of less than four white squares.

5. CARD SHUFFLE

Cards are denoted by the numbers 1-52 stored in an array. They can be converted easily to actual card values when necessary. The cards should be 'shuffled' as follows.

Choose any position in range 1-51 e.g. 17

Place the card in this position in a temporary store.

Shunt all the cards in positions 52 to 18 down to positions 51 to 17

Place the chosen card from the temporary store to position 52.

Deal similarly with the ranges 1-50, 1-49 .. down to 1-2 so that the pack is well shuffled.

Output the result of the shuffle

6. Set up six **DATA** statements each containing a surname, initials and a telephone number (dialling code and local number). Decide on a suitable structure of arrays to store this information and **READ** it into the arrays.

PRINT the data using a separate **FOR** loop and explain how the input format (**DATA**), the internal format (arrays) and output format are not necessarily all the same.

CHAPTER 14 – PROGRAM STRUCTURE

In this chapter we go again over the ground of program structure : loops and decisions or selection. We have tried to present things in as simple a way as possible but SuperBASIC is designed to cope properly with the simple and the complex and all levels in between. Some parts of this chapter are difficult and if you are new to programming you may wish to omit parts. The topics covered are:

Loops
Nested loops
Binary decisions
Multiple decisions

The latter parts of the first section, Loops, get difficult as we show how SuperBASIC copes with problems that other languages simply ignore. Skip these parts if you feel so inclined but the other sections are more straightforward.

LOOPS

In this section we attempt to illustrate the well known problems of handling repetition with simulations of some Wild West scenes. The context may be contrived and trivial but it offers a simple basis for discussion and it illustrates difficulties which arise across the whole range of programming applications.

EXAMPLE 1

A bandit is holed up in the Old School House. The sheriff has six bullets in his gun. Simulate the firing of the six shots.

Program 1

```
100 REMark Western FOR
110 FOR bullets = 1 TO 6
120   PRINT "Take aim"
130   PRINT "Fire shot"
140 END FOR bullets
```

Program 2

```
100 REMark Western REPeat
110 LET bullets = 6
120 REPeat bandit
130 PRINT "Take aim"
140 PRINT "Fire shot"
150 LET bullets = bullets - 1
160 IF bullets = 0 THEN EXIT bandit
170 END REPeat bandit
```

Both these programs produce the same output:

```
Take aim
Fire a shot
```

Is printed six times

If in each program the 6 is changed to any number down to 1 both programs still work as you would expect. But what if the gun is empty before any shots have been fired?

EXAMPLE 2

Suppose that someone has secretly taken all the bullets out of the sheriff's gun. What happens if you simply change the 6 to 0 in each program?

Program 1

```
100 REMark Western FOR Zero Case
110 FOR bullets = 1 to 0
120   PRINT"Take aim"
130   PRINT "Fire a shot"
140 END FOR bullets
```

This works correctly. There is no output. The 'zero case' behaves properly in SuperBASIC

Program 2

```
100 REMark Western REPeat Fails
110 LET bullets = 0
120 REPeat bandit
130   PRINT "Take aim"
140   PRINT "Fire shot"
150   LET bullets = bullets - 1
160   IF bullets = 0 THEN EXIT bandit
170 END REPeat bandit
```

The program fails in two ways:

1.
```
Take aim
Fire a shot
```

   Is printed though there were never any bullets

2. By the time the variable, *bullets,* is tested in line 160 it has the value -1 and it never becomes zero afterwards. The program loops indefinitely. You can cure the infinite looping by re-writing line 160:

```
160 IF bullets < 1 THEN EXIT bandit
```

There is an inherent fault in the programming which does not allow for the possible zero case. This can be corrected by placing the conditional EXIT before the print statements.

Program 3

```
100 REMark Western REPeat Zero Case
110 LET bullets = 0
120 REPeat Bandit
130   IF bullets = 0 THEN EXIT Bandit
140   PRINT "Take aim"
150   PRINT "Fire shot"
160   LET bullets = bullets - 1
170 END REPeat Bandit
```

This program now works properly whatever the initial value of bullets as long as it is a positive whole number or zero. Method 2 corresponds to the **REPEAT.. UNTIL** loop of some languages. Method 3 corresponds to the **WHILE....ENDWHILE** loop of some languages. However the **REPeat.....END REPeat** with **EXIT** is more flexible than either or the combination of both.

If you have used other BASICs you may wonder what has happened to the **NEXT** statement. We will re-introduce it soon but you will see that both loops have a similar structure and both are named.

FOR name =	(opening keyword)	**REPeat** *name*
(statements)	(content)	(statements)
END FOR name	(closing keyword)	**END REPeat** *name*

In addition the **REPeat** loop must normally have an **EXIT** amongst the statements or it will never end.

Note also that the **EXIT** statement causes control to go to the statement which is immediately after the **END** of the loop.

A **NEXT** statement may be placed in a loop. It causes control to go to the statement which is just after the opening keyword **FOR** or **REPeat**. It should be considered as a kind of opposite to the **EXIT** statement. By a curious coincidence the two words, **NEXT** and **EXIT**, both contain EXT. Think of an **EXT**ension to loops and:

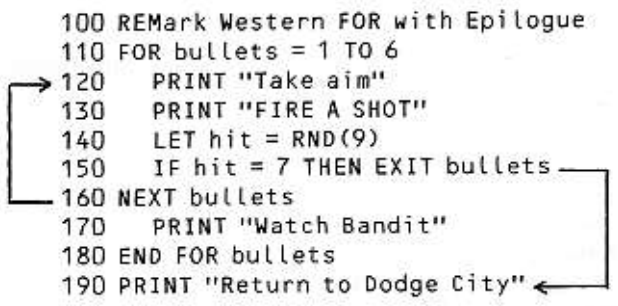
N means "Now start again"
I means "It's ended"

EXAMPLE 3

The situation is the same as in example 1. The sheriff has a gun loaded with six bullets and he is to fire at the bandit but two more conditions apply:

1. If he hits the bandit he stops firing and returns to Dodge City
2. If he runs out of bullets before he hits the bandit, he tells his partner to watch the bandit while he (sheriff) returns to Dodge City

Program 1



In this case, the content between **NEXT** and **END FOR** is a kind of epilogue which is only executed if the FOR loop runs its full course. If there is a premature EXIT the epilogue is not executed.

The same effect can be achieved with a **REPeat** loop though it is not necessarily the best way to do it. However it is worth looking at (perhaps at a second reading) if you want to understand structures which are simple enough to use in simple ways and powerful enough to cope with awkward situations when they arise.

Program 2

```
100 REMark Western REPeat with Epilogue
110 LET bullets = 6
120 REPeat Bandit
130   PRINT "Take aim"
140   PRINT "Fire shot"
150   LET hit = RND(9)
160   IF hit = 7 THEN EXIT Bandit
170   LET bullets = bullets - 1
180   IF bullets <> 0 THEN NEXT Bandit
190   PRINT "Watch Bandit"
200 END REPeat Bandit
210 PRINT "Return to Dodge City"
```

The program works properly as long as the sheriff has at least one bullet at the start. It fails if line 20 reads:

```
110 LET bullets = 0
```

You might think that the sheriff would be a fool to start an enterprise of this kind if he had no bullets at all, and you would be right. We are now discussing how to preserve good structure in the most complex type of situation. We have at least kept the problem context simple: we know what we are trying to do. Complex structural problems usually arise in contexts more difficult than Wild West simulations. But if you really want a solution to the problem which caters for a possible hit, running out of bullets and an epilogue, and also the zero case then add the following line to the above program:

125 IF bullets = 0 THEN PRINT "Watch Bandit" : EXIT bandit

We can conceive of no more complex type of problem than this with a single loop. SuperBASIC can easily handle it if you want it to.

NESTED LOOPS

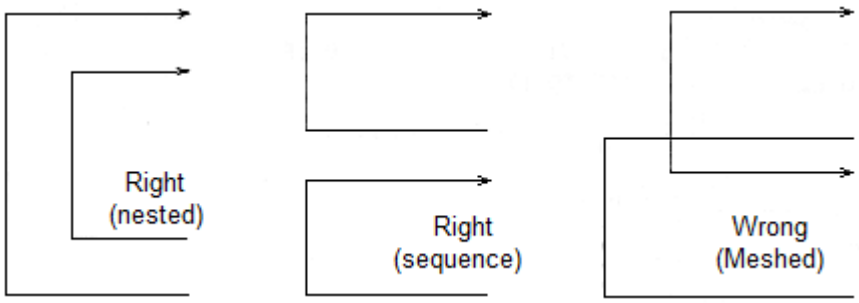
Consider the following FOR loop which PLOTS a row of points of various randomly chosen colours (not black).

```
100 REMark Row of pixels
110 PAPER 0 : CLS
120 LET up = 50
130 FOR across = 20 TO 60
140   INK RND(2 TO 7)
150   POINT across,up
160 END FOR across
```

This program plots a row of points thus:

.....

If you want to get say 51 rows of points you must plot a row for values up from 30 to 80. But you must always observe the rule that a structure can go completely within another or it can go properly around it. It can also follow in sequence, but it cannot 'mesh' with another structure. Books about programming often show how **FOR** loops can be related with a diagram like:



In SuperBASIC the rule applies to all structures. You can solve all problems using them properly. We therefore treat the FOR loop as an entity and design a new program:

```
FOR up = 30 TO 80
  FOR across = 20 TO 60
    INK RND(2 TO 7)
    POINT across,up
  END FOR across
END FOR up
```

When we translate this into a program we are entitled not only to expect it to work but to know what it will do. It will plot a rectangle made up of rows of pixels.

```
100 REMark Rows of pixels
110 PAPER 0 : CLS
120 FOR up = 30 TO 80
130   FOR across = 20 TO 60
140     INK RND(2 TO 7)
150     POINT across,up
160   END FOR across
170 END FOR up
```

Different structures may be nested. Suppose we replace the inner FOR loop of the above program by a REPeat loop. We will terminate the REPeat loop when the zero colour code appears for a selection in the range 0 to 7.

```
100 REMark REPeat in FOR
110 PAPER 0 : CLS
120 FOR up = 30 TO 80
130   LET across = 19
140   REPeat dots
150     LET colour = RND(7)
160     INK colour
170     LET across = across + 1
180     POINT across,up
190     IF colour = 0 THEN EXIT dots
200   END REPeat dots
210 END FOR up
```

Much of the wisdom about program control and structure can be expressed in two rules:

1. Construct your program using only the legitimate structures for loops and decision making.

2. Each structure should be properly related in sequence or wholly within another.

BINARY DECISIONS

The three types of binary decision can be illustrated easily in terms of what to do when when it rains.

Example 1:

```
100 REMark Short form IF
110 LET rain = RND(0 TO 1)
120 IF rain THEN PRINT "Open brolly"
```

Example 2:

```
100 REMark Long form IF. ..END IF
110 LET rain = RND(0 TO 1)
120 IF rain THEN
130   PRINT "Wear coat"
140   PRINT "Open brolly"
150   PRINT "Walk fast"
160 END IF
```

Example 3:

```
100 REMark Long form IF ...ELSE...END IF
110 LET rain = RND(0 TO 1)
120 IF rain THEN
130   PRINT "Take a bus"
140 ELSE
150   PRINT "Walk"
160 END IF
```

AII these are binary decisions. The first two examples are simple : either something happens or it does not. The third is a general binary decision with two distinct possible courses of action, both of which must be defined.

You can omit **THEN** in the long forms if you wish. In the short form you can substitute **:** for **THEN**.

EXAMPLE

Consider a more complex example in which it seems natural to nest binary decisions. This type of nesting can be confusing and you should only do it if it seems the most natural thing to do. Careful attention to layout, particularly indenting, is especially important.

Analyse a piece of text to count the number of vowels, consonants and other characters. Ignore spaces. For simplicity the text is all upper case.

Data:

"COMPUTER HISTORY WAS MADE IN 1984"

Design:

Read in the data
FOR each character:
  IF letter **THEN**
    IF vowel
      increase vowel count
    ELSE
      increase consonant count
    END IF
  ELSE
    IF not space **THEN** increase other count
 END IF
END FOR
PRINT results

```
100 REMark Character Counts
110 RESTORE 290
120 READ text$
130 LET vowels = 0 : cons = 0 : others = 0
140 FOR num = 1 TO LEN(text$)
150   LET ch$ = text$(num)
160   IF ch$ >= "A" AND ch$ <= 'Z'
170     IF ch$ INSTR "AEIOU"
180       LET vowels = vowels + 1
190     ELSE
200       LET cons = cons + 1
210     END IF
220   ELSE
230     IF ch$ <> " " THEN others = others + 1
240   END IF
250 END FOR num
260 PRINT "Vowel count is" ! vowels
270 PRINT "Consonant count is" ! cons
280 PRINT "Other count is" ! others
290 DATA "COMPUTER HISTORY WAS MADE IN 1984"
```

Output

```
Vowel count is 9
Consonant count is 15
Other count is 4
```

MULTIPLE DECISIONS - SELect

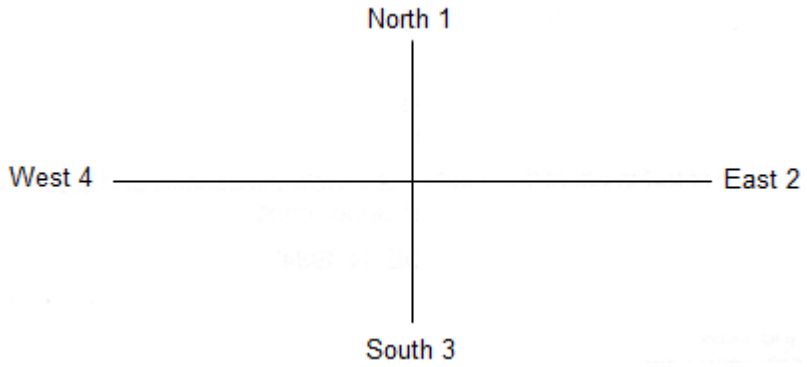
Where there are three or more possible actions and none is dependant on a previous choice the natural structure to use is SELect which enables selection from any number of possibilities.

EXAMPLE

A magic snake grows without limit by adding a section to its front. Each section may be up to twenty units long and may be a new colour or it may remain the same. Each new section must grow in one of the directions North, South, East, or West. The snake starts from the centre of the window.

Method

At any time while the snake is still on the screen you choose a random length and ink colour easily. The direction may be selected by a number 1,2,3 or 4 as shown:



Design:

Select PAPER
Set snake to centre of window
REPeat
  Choose direction, colour length of growth
  FOR unit = 1 to growth
    Make snake grow north, south, east or west
    IF snake is off window THEN EXIT
  END FOR
END REpeat
PRINT end message

Program:

```
100 REMark Magic Snake
110 PAPER 0 : CLS
120 LET across = 50 : up = 50
130 REPeat snake
140   LET direction = RND(1 TO 4) : colour = RND(2 TO 7)
150   LET growth = RND(2 TO 20)
160   INK colour
170   FOR unit = 1 TO growth
180     SELect ON direction
190       ON direction = 1
200         LET up = up + 1
210       ON direction = 2
220         LET across = across + 1
230       ON direction = 3
240         LET up = up - 1
250       ON direction = 4
260         LET across = across - 1
```

```
270     END SELect
280     IF across < 1 OR across > 99 OR up < 1 OR up > 99 : EXIT
snake
290     POINT across,up
300   END FOR unit
310 END REPeat snake
320 PRINT "Snake off edge"
```

The syntax of the **SELect ON** structure also allows for the possibility of selecting on a list of values such as

```
5,6,8,10 TO 13
```

It is also possible to allow for an action to be executed if none of the stated values is found. The full structure is of the form given below.

LONG FORM

SELect ON num
ON num = list of values
  statements
ON num = list of values
  Statements
  -
  -
  -
  -
ON num = **REMAINDER**
  statements
END SELect

where num is any numeric variable and the **REMAINDER** clause is optional.

SHORT FORM

There is a short form of the SELect structure. For example:

```
100 INPUT num
110 SELect ON num = 0 TO 9 : PRINT "digit"
```

will perform as you would expect.

PROBLEMS ON CHAPTER 14

1. Store 10 numbers in an array and perform a 'bubble-sort'. This is done by comparing the first pair and exchanging, if necessary the second pair (second and third numbers), up to the ninth pair (ninth and tenth numbers). The first run of nine comparisons and possible exchanges guarantees that the highest number will reach its correct position. Another eight runs will guarantee eight more correct positions leaving only the lowest number which must be in the only (correct) position left. The simplest form of 'bubble sort' of ten numbers requires nine runs of nine comparisons.

2. Consider ways of speeding up bubblesort, but do not expect that it will ever be very efficient.

3. An auctioneer wishes to sell an old clock and he has instructions to invite a first bid of £50. If no-one bids he can come down to £40, £30, £20, but no lower, in an effort to start the bidding. If no-one bids, the clock is withdrawn from the sale. When the bidding starts, he takes only £5 increases until the final bid is made. If the final bid is £35 (the 'reserve price') or more, the clock is sold. Otherwise it is withdrawn.

Simulate the auction using the equivalent of a six-sided die throw to start the bidding. A 'six' at any of the starting prices will start it off.

When the bidding has started there should be a three out of four chance of a higher bid at each invitation.

4. In a wild west shoot-out the Sheriff has no ammunition and wishes to arrest a gunman camped in a forest. He rides amongst the trees tempting the gunman to fire. He hopes that when six shots have been fired he can rush in and overpower the gunman as he tries to re-load. Simulate the encounter giving the gunman a one-twentieth chance of hitting the Sheriff with each shot. If the Sheriff has not been hit after six shots he will arrest the gunman.

5. The Sheriff's instructions to his Deputy are:

> "If the gun is empty then re-load it and if it ain't then keep on firing until you hit the bandit or he surrenders. If Mexico Pete turns up, get out fast."

Write a program which caters properly for all these situations:

> Whatever happens, return to Dodge City
> If Mexico Pete turns up, return immediately
> If the gun is empty reload it
> If the gun is not empty ask the bandit to surrender.
> If the bandit surrenders, arrest him.
> If he doesn't surrender fire a shot.
> If the bandit is hit, arrest him and fix his wound.

Assume an unlimited supply of ammunition Use a simulated 'twenty-sided die' and let a seven mean 'surrender' and a 'thirteen' mean the bandit is hit.

CHAPTER 15 – PROCEDURES AND FUNCTIONS

In the first part of this chapter we explain the more straightforward features of SuperBASIC's procedures and functions. We do this with very simple examples so that you can understand the working of each feature as it is described. Though the examples are simple and contrived you will appreciate that, once understood, the ideas can be applied in more complex situations where they really matter.

After the first part there is a discussion which attempts to explain 'Why procedures' . If you understand, more or less, up to that point you will be doing well and you should be able to use procedures and functions with increasing effectiveness.

SuperBASIC first allows you to do the simpler things in simple ways and then offers you more if you want it. Extra facilities and some technical matters are explained in the second part of this chapter but you could omit these, certainly at a first reading, and still be in a stronger position than most users of older types of BASIC.

VALUE PARAMETERS

You have seen in previous chapters how a value can be passed to a procedure. Here is another example.

EXAMPLE

In "Chan's Chinese Take-Away" there are just six items on the menu.

Rice Dishes	Sweets
1 prawns	4 ice
2 chicken	5 fritter
3 special	6 lychees

Chan has a simple way of computing prices. He works in pence and the prices are:

for a rice dish 300 + 10 times menu number
for a sweet 12 times menu number

Thus a customer who ate special rice and an ice would pay:

300 + 10 * 3 + 12 * 4 = 378 pence

A procedure, item, accepts a menu number as a value parameter and prints the cost.

Program

```
100 REMark Cost of Dish
110 item 3
120 item 4
130 DEFine PROCedure item(num)
140   IF num <= 3 THEN LET price = 300 + 10*num
150   IF num >= 4 THEN LET price = 12*num
160   PRINT ! price !
170 END DEFine
```

Output

```
330 48
```

In the main program actual parameters 3 and 4 are used. The procedure definition has a formal parameter num, which takes the value passed to it from the main program. Note that the formal parameters must be in brackets, but that actual parameters need not be.

EXAMPLE

Now suppose the working variable, "price", was also used in the main program, meaning something else, say the price of a glass of lager 70p. The following program fails to give the desired result.

```
100 REMark Global price
110 LET price = 70
120 item 3
130 item 4
140 PRINT ! price !
150 DEFine PROCedure item(num)
160   IF num <= 3 THEN LET price = 300 + 10*num
170   IF num >= 4 THEN LET price = 12*num
180   PRINT ! price !
190 END DEFine
```

Output

```
330 48 48
```

The price of the lager has been altered by the procedure. We say that the variable, *price*, is **global** because it can be used anywhere in the program.

Make the procedure variable, *price*, **LOCAL** to the procedure. This means that SuperBASIC will treat it as a special variable accessible only within the procedure. The variable, "price", in the main program will be a different thing even though it has the same name.

```
100 REMark LOCAL price
110 LET price = 70
120 item 3
130 item 4
140 PRINT ! price !
150 DEFine PROCedure item(num)
160   LOCaL price
170   IF num <= 3 THEN LET price = 300 + 10*num
180   IF num >= 4 THEN LET price = 12*num
190   PRINT ! price !
200 END DEFine
```

Output

```
330 48 70
```

This time everything works properly. Line 70 causes the procedure variable, *price* to be internally marked as 'belonging' only to the procedure, *item*. The other variable, *price* is not affected. You can see that local variables are useful things.

EXAMPLE

Local variables are so useful that we automatically make procedure formal parameters local. Though we have not mentioned it before parameters such as *num* in the above programs cannot interfere with main program variables. To prove this we drop the **LOCAL** statement from the above program and use *num* for the price of lager. Because *num* in the procedure is local everything works.

Program

```
100 REMark LOCAL parameter
110 LET num = 70
120 item 3
130 item 4
140 PRINT ! num !
150 DEFine PROCedure item(num)
160   IF num <= 3 THEN LET price = 300 + 10*num
170   IF num >= 4 THEN LET price = 12*num
180   PRINT ! price !
190 END DEFine
```

Output

```
330 48 70
```

VARIABLE PARAMETERS

So far we have only used procedure parameters for passing values to the procedure. But suppose the main program wants the cost of an item to be passed back so that it can compute the total bill. We can do this easily by providing another parameter in the procedure call. This must be a variable because it has to receive a value from the procedure. We therefore call it a variable parameter and it must be matched by a corresponding variable parameter in the procedure definition.

EXAMPLE

Use actual variable parameters, cost_1 and cost_2 to receive the values of the variable price from the procedure. Make the main program compute and print the total bill.

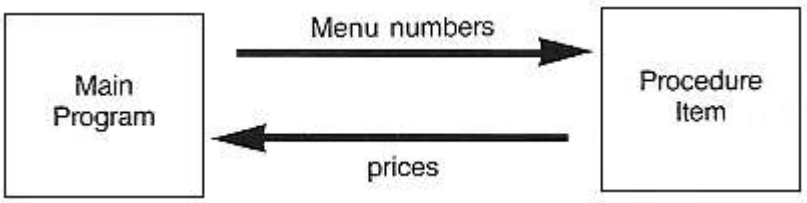
Program

```
100 REMark Variable parameter
110 LET num = 70
120 item 3,cost_1
130 item 4,cost_2
140 LET bill = num + cost_1 + cost_2
150 PRINT bill
160 DEFine PROCedure item(num,price)
170   IF num <= 3 THEN LET price = 300 + 10*num
180   IF num >= 4 THEN LET price = 12*num
190 END DEFine
```

Output

```
448
```

The parameters num and price are both automatically local so there can be no problems. The diagrams show how information passes from main program to procedure and back.



That is enough about procedures and parameters for the present.

FUNCTIONS

You already know how a system function works. For example the function:

```
SQRT(9)
```

computes the value, 3, which is the square root of 9. We say the function returns the value 3. A function, like a procedure, can have one or more parameters, but the distinguishing feature of a function is that it returns exactly one value. This means that you can use it in expressions that you already have. You can type:

```
PRINT 2*SQRT(9)
```

and get the output 6. Thus a function behaves like a procedure with one or more value parameters and exactly one variable parameter holding the returned value: that variable parameter is the function name itself.

The parameters need not be numeric.

```
LEN("string")
```

has a string argument but it returns the numeric value 6.

EXAMPLE

Re write the program of the last section which used price as a variable parameter. Let price be the name of the function.

The value to be returned is defined by the **RETurn** statement as shown.

Program

```
100 REMark FuNction with RETurn
110 LET num = 70
120 LET bill = num + price(3) + price(4)
130 PRINT bill
140 DEFine FuNction price(num)
150   IF num <= 3 THEN RETurn 300 + 10*num
160   IF num >= 4 THEN RETurn 12*num
170 END DEFine
```

Output

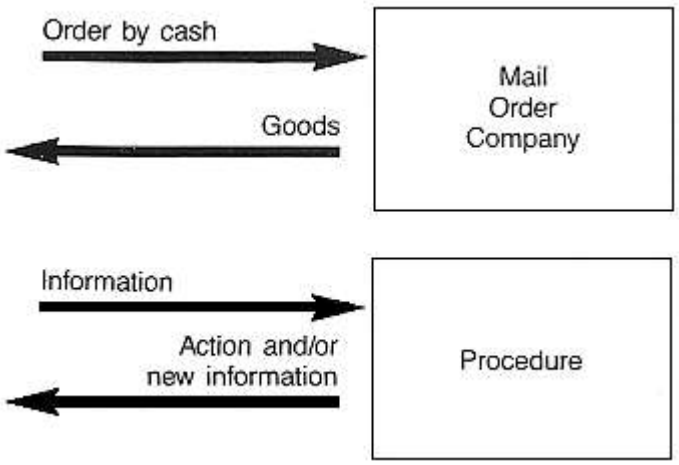
```
 448
```

Notice the simplification in the calling of functions as compared with procedure calls.

WHY PROCEDURES?

The ultimate concept of a procedure is that it should be a 'black box' which receives specific information from 'outside' and performs certain operations which may include sending specific information back to the 'outside: The 'outside' may be the main program or another procedure.

The term 'black box' implies that its internal workings are not important: you only think about what goes in and what comes out. If for example, a procedure uses a variable, count and changes its value, that might affect a variable of the same name in the main program. Think of a mail order company You send them an order and cash: they send you goods. Information is sent to a procedure and it sends back action and/or new information.



You do not want the mail order company to use your name and address or other information for other purposes. That would be an unwanted side-effect. Similarly you do not want a procedure to cause unplanned changes to values of variables used in the main program.

Of course you could make sure that there are no double uses of variable names in a program. That will work up to a point but we have shown in this chapter how to avoid trouble even if you forget what variables have been used in any particular procedure.

A second aim in using procedures is to make a program modular Rather than have one long main program you can break the job down into what Seymour Papert, the inventor of **LOGO**, calls 'Mind-sized bites'. These are the procedures, each one small enough to understand and control easily. They are linked together by the procedure calls in a sequence or hierarchy.

A third aim is to avoid writing the same code twice. Write it once as a procedure and call it twice if necessary. Functions and procedures written for one program can often be directly used, without change, by other programs, and one might create a library of commonly used procedures and functions.

We give below another example which shows how procedures make a program modular.

EXAMPLE

An order is placed for six dishes at Chan's Take Away where the menu is:

Item Number	Dish	Price
1	Prawns	3.50
2	Chicken	2.80
3	Special	3.30

Write procedures for the following tasks.

1. Set up two three-element arrays showing menu, dishes and prices. Use a **DATA** statement.

2. Simulate an order for six randomly chosen dishes using a procedure, choose, and make a tally of the number of times each dish is chosen.

3. Pass the three numbers to a procedure, *waiter*, which passes back the cost of the order to the main program using a parameter *cost*. Procedure waiter calls two other procedures, *compute* and *cook*, which compute the cost and simulate "cooking"

4. The procedure, *cook*, simply prints the number required and the name of each dish.

The main program should call procedures as necessary, get the total cost from procedure, waiter add 10% for a tip, and print the amount of the total bill.

DESIGN

This program illustrates parameter passing in a fairly complex way and we will explain the program step by step before putting it together.

```
100 REMark Procedures
110 RESTORE 490
120 DIM item$(3,7),price(3),dish(3)
130 REMark *** PROGRAM ***
140 LET tip = 0.1
150 set_up
    -
    -
210 DEFine PROCedure set_up
220   FOR k = 1 TO 3
230      READ item$(k)
240      READ price(k)
250   END FOR k
260 END DEFine
    -
    -
    -
490 DATA "Prawns", 3.5, "Chicken", 2.8, "Special" ,3.3
```

The names of menu items and their prices are placed in the arrays *item$* and *price*.

The next step is to choose a menu number for each of the six customers. The tally of the number of each dish required will be kept in the array *dish*.

```
160 choose dish
    -
    -
    -
270 DEFine PROCedure choose(dish)
280   FOR pick = 1 TO 6
290     LET number = RND(1 TO 3)
300     LET dish(number) = dish(number) + 1
310   END FOR pick
320 END DEFine
```

Note that the formal parameter *dish* is both:

local to procedure choose
an array in main program

The three values are passed back to the global array also called dish. These values are then passed to the procedure *waiter*.

```
170 waiter dish, bill
    -
    -
330 DEFine PROCedure waiter (dish, cost)
340   compute dish,cost
350   cook dish
360 END DEFine
```

The waiter passes the information about the number of each dish required to the procedure, *compute*, which computes the cost and returns it.

```
370 DEFine PROCedure compute(dish, total)
380   LET total = 0
390   FOR k = 1 to 3
400     LET total = total + dish(k)*price(k)
410   END FOR k
420 END DEFine
```

The waiter also passes information to the cook who simply prints the number required for each menu item.

```
430 DEFine PROCedure cook(dish)
440   FOR c = 1 TO 3
450     PRINT ! dish(c) ! item$(c) !
460   END FOR c
470 END DEFine
```

Again, the array dish in the procedure cook is local. It receives the information which the procedure uses in its **PRINT** statement.

The complete program is listed below.

```
100 REMark Procedures
110 RESTORE 490
120 DIM item$(3,7),price(3),dish(3)
130 REMark *** PROGRAM ***
140 LET tip = 0.1
150 set_up
160 choose dish
170 waiter dish,bill
180 LET bill = bill + tip*bill
190 PRINT "Total cost is £" ; bill
200 REMark *** PROCEDURE DEFINITIONS ***
210 DEFine PROCedure set_up
220   FOR k = 1 TO 3
230     READ item$(k)
240     READ price(k)
250   END FOR k
260 END DEFine
270 DEFine PROCedure choose(dish)
280   FOR pick = 1 TO 6
290     LET number = RND(1 TO 3)
300     LET dish(number) = dish(number) + 1
310   END FOR pick
320 END DEFine
330 DEFine PROCedure waiter(dish,cost)
340   compute dish,cost
350   cook dish
360 END DEFine
370 DEFine PROCedure compute(dish,total)
380   LET total = 0
390   FOR k = 1 TO 3
400     LET total = total + dish(k)*price(k)
410   END FOR k
420 END DEFine
430 DEFine PROCedure cook(dish)
440   FOR c = 1 TO 3
450     PRINT ! dish(c) ! item$(c)
460   END FOR c
470 END DEFine
480 REMark *** PROGRAM DATA ***
```

```
490 DATA "Prawns",3.5,"Chicken",2.8,"Special",3.3
```

The output depends on the random choice of dishes but the following choice illustrates the pattern, and gives a sample of output.

```
3 Prawns
1 Chicken
2 Special
Total cost is £20.40
```

COMMENT

Obviously the use of procedures and parameters in such a simple program is necessary but imagine that each sub-task might be much more complex. In such a situation the use of procedures would allow a modular build-up of the program with testing at each stage. The above example merely illustrates the main notations and relationships of procedures.

Similarly the next example illustrates the use of functions.

Note that in the previous example the procedures "waiter" and "compute" both return exactly one value. Rewrite the procedures as functions and show any other changes necessary as a consequence.

```
DEFine FuNction waiter(dish)
  cook dish
  RETurn compute(dish)
END DEFine
DEFine FuNction compute(dish)
  LET total = 0
  FOR k = 1 TO 3
    LET total = total + dish(k) * price(k)
  END FOR k
RETurn total
END DEFine
```

The function call to *waiter* also takes a different form

```
LET bill = waiter(dish)
```

This program works as before. Notice that there are fewer parameters though the program structure is similar. That is because the function names are also serving as parameters retuning information to the source of the function call.

EXAMPLE

All the variables used as formal parameters in procedures or functions are 'safe' because they are automatically local. Which variables used in the procedures or functions are not local? What additional statements would be needed to make them local?

Program Changes

The variables *k, pick* and *num* are not local. The necessary changes to make them so are:

```
LOCAL k
LOCAL pick,num
```

TYPELESS PARAMETERS

Formal parameters do not have any type. You may prefer that a variable which handles numbers has the appearance of a numeric variable and which handles strings looks like a string variable, but however you write your parameters they are typeless. To prove it, try the following program.

Program

```
100 REMark Number or word
110 waiter 2
120 waiter "Chicken"
130 DEFine PROCedure waiter(item)
140   PRINT ! item !
150 END DEFine
```

Output

```
2 Chicken
```

The type of the parameter is determined only when the procedure is called and an actual parameter 'arrives'.

SCOPE OF VARIABLES

Consider the following program and try to consider what two numbers will be output.

```
100 REMark scope
110 LET number = 1
120 test
130 DEFine PROCedure test
140   LOCal number
150   LET number = 2
160   PRINT number
170   try
180 END DEFine
190 DEFine PROCedure try
200   PRINT number
210 END DEFine
```

Obviously the first number to be printed will be 2 but is the variable *number* in line 200 global?

The answer is that the value of *number* in line 160 will be carried into the procedure *try*. A variable which is local to a procedure will be the same variable in a second procedure called by the first.

Equally if the procedure *try* is called by the main program, the variable *number* will be the same number in both the main program and procedure, *try*. The implications may seem strange at first but they are logical.

1. The variable number in line 110 is global.

2. The variable number in procedure "test" is definitely local to the procedure.

3. The variable number in procedure "try" 'belongs' to the part of the program which was the last call to it.

We have covered many concepts in this chapter because SuperBASIC functions and procedures are very powerful. However you should not expect to use all these features immediately. Use procedures and functions in simple ways at first. They can be very effective and the power is there if you need it.

PROBLEMS ON CHAPTER 15

1. Six employees are identified by their surnames only. Each employee has a particular pension fund rate expressed as a percentage. The following data represent the total salaries and pension fund rates of the six employees.

Benson	13,800	6.25
Hanson	8,700	6.00
Johnson	10,300	6.25
Robson	15,000	7.00
Thomson	6,200	6.00
Watson	5,100	5.75

   Write procedures to:

   input the data into arrays.
   compute the actual pension fund contributions.
   output the lists of names and computed contributions.

   Link the procedures with a main program calling them in sequence.

2. Write a function *select* with two arguments *range* and *miss*. The function should return a random whole number in the given range but it should not be the value of *miss*.

   Use the function in a program which chooses a random **PAPER** colour and then draws random circles in random **INK** colours so that none is in the colour of **PAPER**.

3. Re-write the solution to exercise 1 so that a function pension takes salary and contribution rate as arguments and returns the computed pension contribution. Use two procedures, one to input the data and one to output the required information using the function *pension*.

4. Write the following:

   a procedure which sets up a 'pack of cards'.

   a procedure which shuffles the cards.

   a function which takes a number as an argument and returns a string value describing the card.

   a procedure which 'deals' and displays four poker hands of five cards each.

   a main program which calls the above procedures.

   (see chapter 16 for discussion of a similar problem)

CHAPTER 16 – SOME TECHNIQUES

In this final chapter we present some applications of concepts and facilities already discussed and we show how some further ideas may be applied.

SIMULATION OF CARD PLAYING

It is easy to store and manipulate "playing cards" by representing them with the numbers 1 to 52. This is how you might convert such a number to the equivalent card. Suppose, for example, that the number 29 appears. You may decide that:

cards 1-13 are hearts
cards 14-26 are clubs
cards 27-39 are diamonds
cards 40 52 are spades

and you will know that 29 means that you have a "diamond". You can program the QL to do this with:

```
LET suit = (card-1) DIV 13
```

This will produce a value in the range 0 to 3 which you can use to cause the appropriate suit to be printed. The value can be reduced to the range 1 to 13 by writing:

```
LET value = card MOD 13
IF value = 0 THEN LET value = 13
```

Program

The numbers 1 to 13 can be made to print Ace, 2, 3... Jack, Queen, King, or if you prefer it, such phrases as "two of hearts" can be printed. The following program will print the name of the card corresponding to your input number.

```
100 REMark Cards
110 DIM suitname$(4,8),cardval$(13,5)
120 LET f$ = " of"
130 set_up
140 REPeat cards
150   INPUT "Enter a card number 1-52:" ! card
160   IF card <1 OR card> 52 THEN EXIT cards
170   LET suit = (card-1) DIV 13
180   LET value = card MOD 13
190   IF value = 0 THEN LET value = 13
200   PRINT cardval$(value) ! f$ ! suitname$(suit)
210 END REPeat cards
220 DEFine PROCedure set_up
230   FOR s = 1 TO 4 : READ suitname$(s)
240   FOR v = 1 TO 13 : READ cardval$(v)
250 END DEFine
260 DATA "hearts","clubs","diamonds","spades"
270 DATA "Ace","Two","Three","Four","Five","Six","Seven"
280 DATA "Eight","Nine","Ten","Jack","Queen","King"
```

Input and Output

```
13
King of hearts
49
Ten of spades
27
```

```
Ace of diamonds
0
```

COMMENT

Notice the use of **DATA** statements to hold a permanent file of data which the program always uses. The other data which changes each time the program runs is entered through an **INPUT** statement. If the input data was known before running the program it would be equally correct to use another **READ** and more **DATA** statements. This would give better control.

SEQUENTIAL DATA FILES

The following program will establish a file of one hundred numbers.

```
100 REMark Number File
110 OPEN NEW #6,mdv1_numbers
120 FOR num = 1 TO 100
130   PRINT #6,num
140 END FOR num
150 CLOSE #6
```

Numeric File

After running the program check that the filename 'numbers' is in the directory by typing:

```
DIR mdv1_numbers
```

You can get a view of the file without any special formatting by copying from Microdrive to screen:

```
COPY mdv1_numbers to scr
```

You can also use the following program to read the file and display its records on the screen.

```
100 REMark Read File
110 OPEN_IN #6,mdv1_numbers
120 FOR num = 1 TO 100
130   INPUT #6,item
140   PRINT ! item !
150 END FOR num
160 CLOSE #6
```

If you wish you can alter the program to get the output in a different form.

Character File

In a similar fashion the following programs will set up a file of one hundred randomly selected letters and read them back.

```
100 REMark Letter File
110 OPEN NEW #6,mdv1_chfile
120 FOR num = 1 TO 100
130   LET ch$ = CHR$(RND(65 TO 90))
140   PRINT #6,ch$
150 END FOR num
160 CLOSE #6

100 REMark Get Letters
110 OPEN IN #6,mdv1_chfile
120 FOR num = 1 TO 100
130   INPUT #6,item$
140   PRINT ! item$ !
```

```
150 END FOR num
160 CLOSE #6
```

Suppose that you wish to set up a simple file of names and telephone numbers.

```
RON       678462
GEOFF     896487
ZOE       249386
BEN       584621
MEG       482349
CATH      438975
WENDY     982387
```

The following program will do it.

```
100 REMark Phone numbers
110 OPEN NEW #6,mdv1_phone
120 FOR record = 1 TO 7
130   INPUT name$,num$
140   PRINT #6;name$;num$
150 END FOR record
160 CLOSE #6
```

Type RUN and enter a name followed by the ENTER key and a number followed by the ENTER key. Repeat this seven times.

Notice that the data is 'buffered'. It is stored internally until the system is ready to transfer a batch to the Microdrive. The Microdrive is only accessed once, as you can tell from looking and listening.

COPY A FILE

Once a file is established, it should be copied immediately as a back-up. To do this type:

```
COPY mdv1_phone TO mdv2_phone
```

READ A FILE

You need to be certain that the file exists in a correct form so you should read it back from a Microdrive and display it on the screen. You can do this easily using:

```
COPY mdv2_phone TO scr
```

The output to the screen will not provide spaces automatically between the name and the number but it will provide a 'newline' at the end of each record. The output will be:

```
RON678462
GEOFF896487
ZOE249386
BEN584621
MEG482349
CATH438975
WENDY982387
```

You can get a more controlled presentation of the data with the following program.

```
100 REMark Read Phone Numbers
110 OPEN_IN #5,mdv1_phone
120 FOR record = 1 TO 7
130   INPUT #5,rec$
140   PRINT,rec$
```

```
150 END FOR record
160 CLOSE #5
```

The data is printed, as before, but this time each pair of fields is held in the variable *rec$* before being printed on the screen. You have the opportunity to manipulate it into any desired form.

Note that more than one string variable may be used at the file creation stage with **INPUT** and **PRINT** but the whole record so created may be retrieved from the Microdrive file with a single string variable (*rec$* in the above example).

AN INSERTION SORT

The following colours are available in the low resolution screen mode (in code number order 0-7).

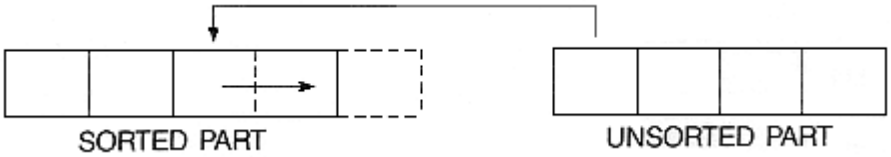
black blue red magenta green cyan yellow white

EXAMPLE

Write a program to sort the colours into alphabetical order using an "insertion" sort.

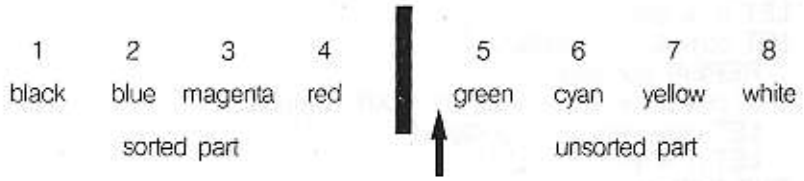
Method

We place the eight colours in an array *colour$* which we divide into two parts:



We take the leftmost item of the unsorted part and compare it with each item, from right to left, in the sorted part until we find its right place. As we compare we shuffle the sorted items to the right so that when we find the right place to insert we can do so immediately without further shuffling.

Suppose we have reached the point where four items are sorted and we now focus on green, the leftmost item in the unsorted part.

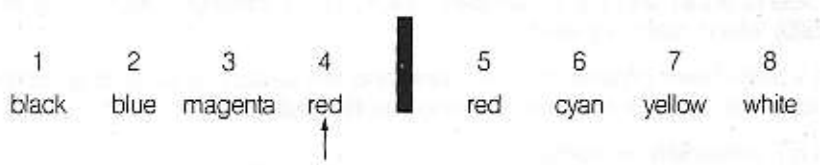


1. We place green in the variable, *comp$*, and set a variable, *p*, to 5.

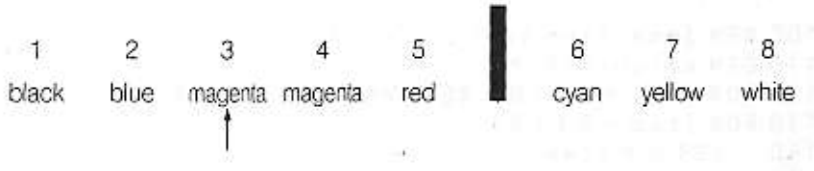
2. The variable, *p*, will eventually indicate where we think green should go. When w know that green should move left, then we decrease the value of *p*.

3. We compare green with red. If green is greater than (nearer to Z) or equal to red we exit and green stays where it is.

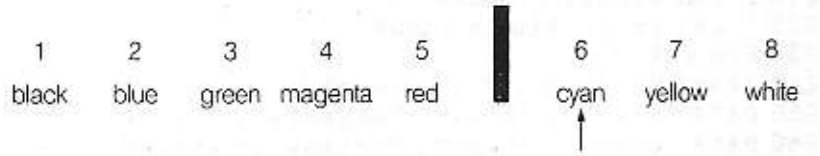
   Otherwise we copy red in to position 5 thus and decrease the value of *p* thus:



4. We now repeat the process but this time we are comparing green with magenta and we get:



5. Finally we move left again comparing green with blue. This time there is no need to move or change anything. We exit from the loop and place green in position 3. We are then ready to focus on the sixth item, cyan.



PROBLEM ANALYSIS

1. We will first store the *colour$* in an array *colour$(8)* and use:

comp$ the current colour being compared
p to point at the position where we think the colour in *comp$* might go.

2. A **FOR** loop will focus attention on positions 2 to 8 in turn (a single item is already sorted).

3. A **REPeat** loop will allow comparisons until we find where the *comp$* value actually goes.

```
REPeat compare
  IF comp$ need go no further left EXIT
  copy a colour into the position on its right
  and decrease p
END REPeat compare
```

4. After **EXIT** from the **REPeat** loop the colour in *comp$* is placed in position *p* and the **FOR** loop continues.

Program Design

```
1 Declare array colour$
2 Read colours into the array
3 FOR item = 2 TO 8
    LET p = item
    LET comp$ = colour$(p)
      REPEAT compare
        IF comp$ > = colour$(p-1) : EXIT compare
        LET colour$(p) = colour$(p-1)
        LET p = p - 1
      END REPeat compare
      LET colour$(p) = comp$
    END FOR item
4 PRINT sorted array colour$
5 DATA
```

Further testing reveals a fault. It arises very easily if we have data in which the first item is not in its correct position at the start. A simple change of initial data to:

red black blue magenta green cyan yellow white

reveals the problem. We compare black with red and decrease *p* to the value, 1. We come round again and try to compare black with a variable *colour$(p-1)* which is colour$(0) which does not exist.

This is a well-known problem in computing and the solution is to "post a sentinel" on the end of the array. Just before entering the **REPeat** loop we need:

```
LET colour$(0) = comp$
```

Fortunately SuperBASIC allows zero subscripts, otherwise the problem would have to be solved at the expense of readability.

MODIFIED PROGRAM

```
100 REM Insertion Sort
110 DIM colour$(8,7)
120 FOR item = 1 TO 8 : READ colour$(item)
130 FOR item = 2 TO 8
140   LET p=item
150   LET comp$ = colour$(p)
160   LET colour$(0) = comp$
170   REPeat compare
180     IF comp$ >= colour$(p-1) : EXIT compare
190     LET colour$(p) = colour$(p-1)
200     LET p = p-1
210   END REPeat compare
220   LET colour$(p) = comp$
230 END FOR item
240 PRINT"Sorted..." ! colour$
250 DATA "black","blue","magenta","red"
260 DATA "green","cyan","yellow","white"
```

COMMENT

1. The program works well. It has been tested with awkward data:

   A A A A A A A
   B A B A B A B
   A B A B A B A
   B C D E F G H
   G F E D C B A

2. An insertion sort is not particularly fast, but it can be useful for adding a few items to an already sorted list. It is sometimes convenient to allow modest amounts of time frequently to keep items in order rather than a substantial amount of time less frequently to do a complete re-sorting.

You now have enough background knowledge to follow a development of the handling of the file of seven names and telephone numbers.

SORTING A MICRODRIVE FILE

In order to sort the file 'phone' into alphabetical order of names we must read it into an internal array, sort it, and then create a new file which will be in alphabetical order of names.

It is never good practice to delete a file before its replacement is clearly established and proven correct. You should therefore copy the file first, as security using a different name. The required processes are as follows:

1. Copy the file 'phone' to 'phone_temp'
2. Read the file 'phone' into an array

3. Sort the array.
4. Pause to check that everything is in order
5. Delete file 'phone'.
6. Create new file 'phone'.

This is all the program needs to do but the new file should be immediately checked using:

```
COPY mdv1_phone TO scr
```

Any further necessary checks should be carried out then:

```
DELETE mdv2 phone
COPY mdv1_phone TO mdv2_phone
COPY mdv1_phone TO scr
DELETE mdv1_phone_temp
```

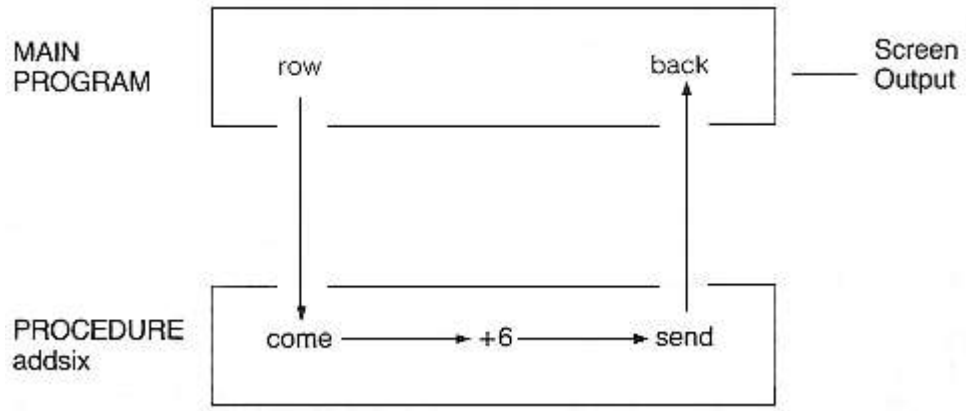
The above operations complete the process of substituting a sorted file for the original unsorted one in both master and back-up files.

ARRAY PARAMETERS

In the following program we illustrate the passing of complete arrays between main program and procedure. The data passes in both directions.

In line 40 the array *row* holding the numbers 1, 2, 3 is passed to the procedure, *addsix*. The parameter come, receives the incoming data and the procedure adds six to each element. The array parameter, *send*, at this point holds the numbers 7,8,9.

These numbers are passed back to the main program to become the values of array, *black*. The values are printed to prove that the data has moved as required.



Program

```
100 REMark Pass Arrays
110 DIM row(3),back(3)
120 FOR k = 1 TO 3 : LET row(k) = k
130 addsix row,back
140 FOR k = 1 TO 3 : PRINT ! back(k) !
150 DEFine PROCedure addsix(come,send)
160   FOR k = 1 TO 3 : LET send(k) = come(k) + 6
170 END DEFine

789
```

Output

The following procedure receives an array containing data to be sorted. The zero element will contain the number of items. Note that it does not matter whether the array is numeric or string. The principle of coercion will change string to numeric data if necessary.

A second point of interest is that the array element, *come(0)*, is used for two purposes:

it carries the number of items to be sorted
it is used to hold the item currently being placed.

```
100 DEFine PROCedure sort(come,send)
110   LET num = come(0)
120   FOR item = 2 TO num
130     LET p = item
140     LET come(0) = come(p)
150     REPeat compare
160       IF come(0) >= come(p-1) : EXIT compare
170       LET come(p) = come(p-1)
180       LET p = p - 1
190     END REPeat compare
200     LET come(p) = come(0)
210   END FOR item
220   FOR k = 1 TO 7 : send(k) = come(k)
230 END DEFine
```

The following additional lines will test the sort procedure. First type **AUTO 10** to start the line numbers from 10 onwards.

```
10 REMark Test Sort
20 DIM row$(7,3),back$(7,3)
30 LET row$(0) = 7
40 FOR k = 1 TO 7 : READ row$(k)
50 sort row$,back$
60 PRINT ! back$ !
70 DATA "EEL","DOG","ANT","GNU","CAT","BUG","FOX"
```

Output

```
ANT BUG CAT DOG EEL FOX GNU
```

COMMENT

This program illustrates how easily you can handle arrays in SuperBASIC. All you have to do is use the array names for passing them as parameters or for printing the whole array. Once the procedure is saved you can use **MERGE mdv1_sort** to add it to a program in main memory.

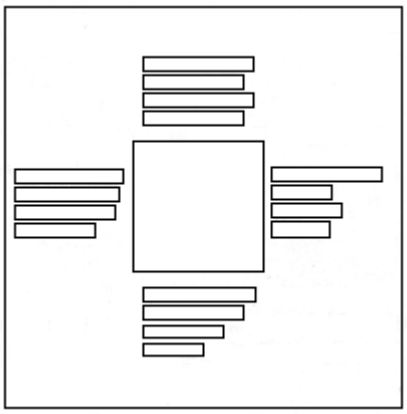
You now have enough understanding of techniques and syntax to handle a more complex screen layout. Suppose you wish to represent the hands of four card players. A hand can be represented by something like:

```
H: A 3 7 Q
C: 5 9 J
D: 6 10 K
S: 2 4 Q
```

To help the presentation the Hearts and Diamonds will be printed in red and the Clubs and Spades in black. A suitable **STRIP** colour might be white. The general background could be green and a table may be a colour obtained by mixing two colours.

METHOD

Since a substantial amount of character printing is involved it is best to start planning in terms of the pixel screen. You can see that you need to provide for twelve lines of characters with some space between lines and a total screen height of 256 pixels.



It is useful to recall that the possible character heights are 10 pixels or 20 pixels. It is obvious that the 10 pixel height must be used to allow space for a proper layout.

The number of character positions across the screen must be estimated. If we adopt the convention of "T" for ten instead of "10" all card values can be represented as a single character. Suppose that we also allow a maximum of eight cards of the same suit as a first approach. We can reconsider the problem again if necessary That would require a total of 10 characters for each hand. The across requirement is therefore:

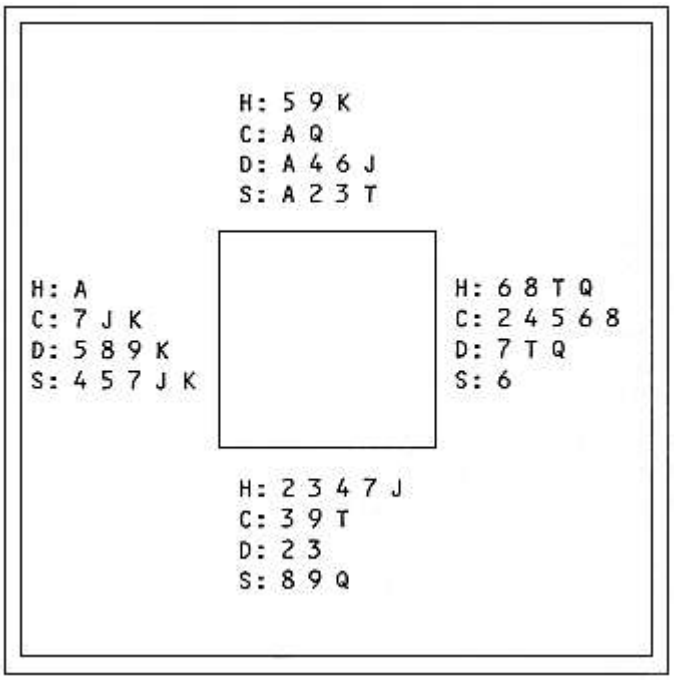
west hand + table width + east hand

Allowing a space between characters that would be:

20 + table width + 20

The decision now depends on which screen mode you choose. The 256 mode will cope with the problem, as you will see later, but first we will work in 512 pixel mode. The smallest character width is six pixels which would give a total of 240 pixels + table width. The diagram will have some balance if we have a table width of about half of 240.

We should therefore experiment with a table width of about 120 pixels which may be adjusted. A little testing produced the layout shown.



WINDOW	440 x 220 at 35,15 Green with black border of 10 units	
TABLE	100 x 60 at 150,60 Chequerboard stipple of red and green	
HANDS	Room for at least eight card symbols Initial cursor positions are:	
	north	150,10
	east	260,60
	south	150,130
	west	30,60
CHARACTER SIZE NUMBER OF PIXELS	Standard for 512 mode between lines is 12	
CHARACTER COLOUR	White	
CHARACTER STRIP	Red for Hearts and Diamonds Black for Clubs and Spades	

VARIABLES

card(52)	stores card numbers
sort(13)	used to sort each hand
tok$(4,2)	stores tokens H:, C:, D:, S:
kmcmh	working loop variables
ran	random position for card exchange
temp	used in card exchange
item	card to be inserted in sort
dart	pointer to find position in sort
comp	hold card number in sort
inc	pixel increment in card rows
seat	current 'deal' position

ac,dn	cursor position for characters
row	current row for characters
lin$	builds up row of characters
max	highest card number
p	points to card position
n	current number of card

PROCEDURES

Shuffle	shuffles 52 cards
Split	splits cards into four hands and calls sortem to sort each hand
Sortem	sorts 13 cards in ascending order
Layout	provides background colour border and table
Printem	prints each line of card symbols
Getline	gets one row of cards and converts numbers into the symbols A,2,3,4,5,6,7,8,9,T,J,Q,K

PROGRAM DESIGN OUTLINE

1. Declare arrays, pick up 'tokens' and place 52 numbers in array card.
2. Shuffle cards.
3. Split into 4 hands and sort each.
4. OPEN screen window.
5. Fix the screen layout.
6. Print the four hands.
7. CLOSE the screen window.

```
100 DIM card(52),sort(13),tok$(4,2)
110 FOR k = 1 TO 4 : READ tok$(k)
120 FOR k = 1 TO 52 : LET card(k) = k
130 shuffle
140 split
150 OPEN #6,scr_440x220a35x15
160 layout
170 printem
180 CLOSE #6
190 DEFine PROCedure shuffle
200   FOR c = 52 TO 3 STEP -1
210     LET ran = RND(1 to c-1)
220     LET temp = card(c)
230     LET card(c) = card(ran)
240     LET card(ran) = temp
250   END FOR c
260 END DEFine
270 DEFine PROCedure split
280   FOR h = 1 TO 4
290     FOR c = 1 TO 13
300       LET sort(c) = card((h-1)*13+c)
310     END FOR c
320     sortem
330     FOR c = 1 TO 13
340       LET card((h-1)*13+c) = sort(c)
350     END FOR c
360   END FOR h
370 END DEFine
380 DEFine PROCedure sortem
390   FOR item = 2 TO 13
400     LET dart = item
410     LET comp = sort(dart)
420     LET sort(0) = comp
```

```
430     REPeat compare
440       IF comp >= sort(dart-1) : EXIT compare
450       LET sort(dart) = sort(dart-1)
460       LET dart = dart - 1
470     END REPeat compare
480     LET sort(dart) = comp
490   END FOR item
500 END DEFine
510 DEFine PROCedure layout
520   PAPER #6,4 : CLS #6
530   BORDER #6,10,0
540   BLOCK #6,100,60,150,60,2,4
550 END DEFine
560 DEFine PROCedure printem
570   LET inc = 12 : INK #6,7
580   LET p = 0
590   FOR seat = 1 TO 4
600     READ ac,dn
610     FOR row = 1 TO 4
620       getline
630       CURSOR #6,ac,dn
640       PRINT #6,1in$
650       LET dn = dn + inc
660     END FOR row
670   END FOR seat
680 END DEFine
690 DEFine PROCedure getline
700   IF row MOD 2 = 0 THEN STRIP #6,0
710   IF row MOD 2 = 1 THEN STRIP #6,2
720   LET lin$ = tok$(row)
730   LET max = row*13
740   REPeat one_suit
750     LET p = p + 1
760     LET n = card(p)
770     IF n >max THEN p = p-1 : EXIT one_suit
780     LET n = n MOD 13
790     IF n = 0 THEN n = 13
800     IF n = 1 : LET ch$ = "A"
810     IF n >= 2 AND n <= 9 : LET ch$ = n
820     IF n = 10 : LET ch$ = "T"
830     IF n = 11 : LET ch$ = "J"
840     IF n = 12 : LET ch$ = "Q"
850     IF n = 13 : LET ch$ = "K"
860     LET lin$ = lin$ & " " & ch$
870     IF p = 52 : EXIT one_suit
880     IF card(p)>card(p+1) : EXIT one_suit
890   END REPeat one_suit
900 END DEFine
910 DATA "H:","C:","D:","S:"
920 DATA 150,10,260,60,150,130,30,60
```

COMMENT

The program works in the 256 mode. But the various lines of card symbols may overlap the "table" or overflow at the edge of the window. A simple change in procedure "getline" from:

```
860 LET lin$ = lin$ & " " & ch$
```

to

```
860 LET lin$ = lin$ & ch$
```

will correct this. The spaces between characters disappear but the larger sized characters enable the rows to be easily readable. The program thus works well in either graphics mode.

CONCLUSION

We have tried to show how you can use SuperBASIC to solve problems. We have shown how simple tasks can be performed in simple ways. When the task is inherently complex, like manipulating arrays or designing screen graphics, SuperBASIC enables it to be handled efficiently with maximum possible clarity.

If you were a beginner and you have worked through a fair proportion of this guide you will have started well on the road to good programming. If you were already experienced, we hope that you will appreciate and exploit the extra features offered by SuperBASIC.

So enormous is the range of tasks which can be done with SuperBASIC that we have only been able to touch a fraction of them in this guide. We cannot guess at which of the thousands of possibilities you will attempt, but we hope that you will find them fruitful, stimulating and fun.

17 - ANSWERS TO SELF TESTS

ANSWERS TO SELF TEST ON CHAPTER 1

1. Use the BREAK sequence to abandon a running program because:
   a) something is wrong and you do not understand it
   b) it is longer of interest
   c) any other problem (three points)

2. The RESET button is on the right hand side of the computer.

3. The effect of the RESET button is rather like switching the computer off and on again.

4. The **SHIFT** key:
   a) is only effective while you hold it down whereas the **CAPS LOCK** key stays effective after you have pressed it. (one point)
   b) The **SHIFT** key affects all the letter digit and symbol keys, but the **CAPS LOCK** key affects only letters. (one point)

5. The CTRL ← (CTRL left arrow) keys delete the previous character just left of the cursor.

6. The ⏎ (ENTER) key causes a message or instruction to be entered for action by the computer.

7. We use ⏎ for the ENTER key.

8. CLS ⏎ causes part of the screen to be cleared.

9. RUN ⏎ causes a stored program to be executed.

10. LIST ⏎ causes a stored program to be displayed on the screen.

11. NEW ⏎ clears the main memory ready for a new program.

12. Keywords of SuperBASIC are recognised in upper or lower case.

13. The part of a keyword displayed in upper case is the allowed abbreviation.

CHECK YOUR SCORE

14 to 16 is very good. Carry on reading.

12 or 13 is good, but re-read some parts of chapter one.

10 or 11 is fair, but re-read some parts of chapter one and do the test again.

Under 10. You should work carefully through chapter one again and repeat the test.

ANSWERS TO SELF TEST ON CHAPTER 2

1. An internal number store is like a pigeon hole which you can name and put numbers into.

2. A **LET** statement which uses a particular name for the first time will cause a pigeon hole to be created and named, for example
   LET count = 1 (1 point)
   A **READ** statement which uses a name for the first time will have the same effect, for example:
   READ count (1 point)

3. You can find the value of a pigeon hole with a **PRINT** statement.

4. The technical name for a pigeon hole is 'variable' because its values can vary as a program runs.

5. A variable gets its first value when it is first used in a **LET** statement, **INPUT** statement or **READ** statement.

6. A change in the value of a variable is usually caused by the execution of a **LET** statement.

7. The = sign in a **LET** statement represents an operation:
   'Evaluate whatever is on the right hand side and place it in the pigeon hole named on the left hand side: that is 'Let the left hand side become equal to the right hand side'.

8. An unnumbered statement is executed immediately.

9. A numbered statement is not executed immediately. It is stored.

10. The quotes in a **PRINT** statement enclose text which is to be printed.

11. When quotes are not used you are printing out the value of a variable.

12. An **INPUT** statement makes the program pause so that you can type data at the keyboard.

13. **DATA** statements are never executed.

14. They are used to provide values for the variables in **READ** statements.

15. The technical word for the name of a pigeon hole is 'identifier'.

16. Example answers:
    i. day
    ii. day_23
    iii. day_of_week (3 points)

17. The space bar is especially important for putting spaces after or before keywords so that they cannot be taken as identifiers (names) chosen by the user.

18. Freely chosen identifiers are important because they help you to make programs easier to understand. Such programs are less prone to errors and more adaptable.

CHECK YOUR SCORE

18 to 21 is very good. Carry on reading.

16 or 17 good but re-read some parts of chapter two.

14 or 15 fair, but re-read some parts of chapter two and do the test again.

Under 14 you should work carefuly through chapter two again and repeat the test.

ANSWERS TO SELF TEST ON CHAPTER 3

1. A pixel is the smallest area of light that can be displayed on the screen.

2. There are 256 pixel positions across the low resolution mode.

3. There are 256 pixel positions from top to bottom in the low resolution mode.

4. An address is determined by.

   the up value, 0 to 100
   the across value, 0 to a number computed by the system

5. There are eight colours available in the low resolution mode including black and white.

6.
   1. **LINE** draws a line, e.g. **LINE a,b TO x,y**

   2. **INK** selects a colour for drawing, e.g. **INK 5**

   3. **PAPER** selects a background colour e.g. **PAPER 7**

   4. **BORDER** draws a border, e.g. **BORDER 1,5**

7. **REPeat** name....**END REPeat** name.

8. A **REPeat** loop terminates when an '**EXIT** name' statement is executed.

9. Loops in SuperBASIC have names so that it is possible to **EXIT** from them in a straightforward way. It is not necessary to work out line numbers in advance.

CHECK YOUR SCORE

11 to 13 is very good. Carry on reading.

8 to 10 is good but re-read some parts of chapter three.

6 or 7 is fair but re-read some parts of chapter three and do the test again.

Under 6. You should work carefully through chapter three again and repeat the test.

ANSWERS TO SELF TEST ON CHAPTER 4

1. A character string is a sequence of characters such as letters, digits or other symbols.

2. The term, 'character string', is often abbreviated to 'string'.

3. A string variable name always ends with **$**.

4. Names such as *word$* are sometimes pronounced 'worddollar'

5. The keyword **LEN** will find the length or number of characters in a string. For example, if the variable *meat$* has the value 'steak' then the statement:

```
PRINT LEN(meat$)
```

   will output 5.

6. The symbol for joining two strings is **&**.

7. The limits of a string may be defined by quotes or apostrophes.

8. The quotes are not part of the actual string and are not stored.

9. The function is **CHR$**. You must use it with brackets as in **CHR$(66)** or with brackets as in **CHR$(RND(65 TO 67))**.

10. You generate random letters with statements like:

```
lettercode = RND(65 TO 90)
PRINT CHR$(lettercode)
```

CHECK YOUR SCORE

9 or 10 is very good. Carry on reading.

7 or 8 is good but re-read some parts of chapter four

5 or 6 is fair but re-read some parts of chapter four and do the test again.

Under 5 You should work carefully through chapter four again and repeat the test.

ANSWERS TO SELF TEST ON CHAPTER 5

1. Lower case letters for variable names or loop names contrast with the keywords which are at least partly displayed in upper case.

2. Indenting reveals clearly what is the extent and content of loops (and other structures).

3. Identifiers (names) should normally be chosen so that they mean something, for example, *count* or *word$* rather than *C* or *W$*

4. You can edit a stored program by:

   replacing a line
   inserting a line
   deleting a line (three points)

5. The **ENTER** key must be used to enter a command or program line.

6. The word **NEW** will wipe out the previous SuperBASIC program in the QL and will ensure that a new program which you enter will not be merged with an old one.

7. If you wish a line to be stored as part of a program then you must use a line number.

8. The word **RUN** followed by ↵ will cause a program to execute.

9. The word **REMark** enables you to put into a program information which is ignored at execution time.

10. The keywords **SAVE** and **LOAD** enable programs to be stored on and retrieved from cartridges.(two points).

CHECK YOUR SCORE

12 to 14 is very good. Carry on reading.

10 or 11 is good but re-read some parts of chapter five.

8 or 9 is fair but re-read some parts of chapter five and do the test again.

Under 8 You should re-read chapter five carefully and do the test again.

ANSWERS TO SELF TEST ON CHAPTER 6

1. It is not easy to think of many different variable names for storing the data. If you can think of enough names, every one has to be written in a **LET** statement or a **READ** statement if you do not use arrays.

2. A number called the subscript, is part of an array variable name. All the variables in an array share one name but each has a different subscript.

3. You must 'declare' an array giving its size (dimension) in a **DIM** statement usually placed near the beginning of a program before the declared array is used.

4. The distinguishing number of an array variable is called the subscript.

5. Houses in a street share the same street name but each has its own number.

   Beds in a hospital ward may share the name of the ward but each bed may be numbered.

   Cells in a prison block may have a common block name but a different number.

   Holes on a golf course, e.g. the fifth hole at Royal Birkdale.

6. A **FOR** loop terminates when the process corresponding to the last value of the loop variable has been completed.

7. A **FOR** loop's name is also the name of the variable which controls the loop.

8. The two phrases for this variable are 'loop variable' or 'control variable'.

9. The values of a loop variable may be used as subscripts for array variable names. Thus, as the loop proceeds, each array variable is 'visited' once.

10. Both **FOR** loops and **REPeat** loops:

    a. have an opening keyword:
       REPeat , FOR
    b. have a closing statement:
       END REPeat name, END FOR name
    c. have a loop name.
       Only the **FOR** loop has
    d. a loop variable or control variable. (four points)

CHECK YOUR SCORE

This test is more searching than the previous ones.

15 or 16 is excellent. Carry on reading.

13 or 14 is very good but think a bit more about some of the ideas. Look at programs to see how they work.

11 or 12 is good but re-read some parts of chapter six.

8 to 10 is fair but re-read some parts of chapter six and do the test again.

Under 8 You should re-read chapter six carefully and do the test again.

ANSWERS TO SELF TEST ON CHAPTER 7

1. We normally break down large or complex jobs into smaller tasks until they are small enough to be completed.

2. This principle can be applied in programming by breaking the total job down and writing a procedure for each task.

3. A simple procedure is:

   a separate block of code
   properly named. (two points)

4. A procedure call ensures that:

   the procedure is activated
   control returns to just after the calling point. (two points)

5. Procedure names can be used in a main program before the procedures have been written. This enables you to think about the whole job and get an overview without worrying about the detail.

6. If you write a procedure definition before using its name you can test it and then when it works properly forget the details. You need only remember its name and roughly what it does.

7. A programmer who can write up to thirty line programs can break down a complex task into procedures in such a way that none is more than thirty lines and most are much less. In this way he need only worry about one bit of the job at a time.

8. The use of a procedure would save memory space if it is necessary to call it more than once from different parts of a program. The definition of a procedure only occurs once but it can be called as often as necessary.

9. A main program can place information in 'pigeon-holes' by means of LET or READ statements. These 'pigeon holes' can be accessed by the procedure. Thus the procedure uses information originally set up by the main program.

   A second method is to use parameters in the procedure call. These values are passed to variables in the procedure definition which then uses them as necessary.

10. An actual parameter is the actual value passed from a procedure all in a main program to a procedure.

11. A formal parameter is a variable in a procedure definition which receives the value passed to the procedure by the main program.

CHECK YOUR SCORE

This is a searching test. You may need more experience of using procedures before the ideas can be fully appreciated. But they are very powerful and, when understood, extremely helpful ideas. They are worth whatever effort is necessary.

12 to 14 excellent. Read on with confidence.

10 or 11 very good. Just check again on certain points.

8 or 9 good but re-read some parts of chapter seven.

6 or 7 fair but re-read some parts of chapter seven. Work carefully through the programs writing down all changes in variable values. Then do the test again.

Under 6 read chapter seven again. Take it slowly working all the programs. These ideas may not be easy but they are worth the effort. When you are ready, take the test again.