VIC 20

COLOUR COMPUTER

# AN INTRODUCTION TO BASIC:~PART 2

THE COMPREHENSIVE TEACH YOURSELF PROGRAMMING SERIES FOR VIC 20

by Andrew Colin


commodore
COMPUTER

# CONTENTS

This course is Part 2 of a series designed to help you learn about every aspect of programming the Commodore VIC 20 computer. It builds upon the principles covered in Part 1 of this series to give you all the knowledge necessary to write good, well designed BASIC programs on your VIC computer. The course has two constituent parts:

1. A self-study text divided into 10 lessons or 'units', each of which deals with an important aspect of programming.

2. Two cassette tapes containing a collection of VIC programs, which will help you study the units.

## CONTENTS LIST

# INTRODUCTION

Welcome to the second part of the VIC BASIC course. The layout of the book and tapes will be familiar to you since the course is a direct continuation of VIC BASIC Part 1. The units have been numbered consecutively from 16 upward to emphasise the arrangement.

At first, you'll find the units quite similar to those you have already studied but as you work through the book they will get longer and perhaps a little harder. We shall be doing advanced applications of BASIC including arrays, the manipulation of strings of characters, animated games, and the use of cassette tapes as backing stores. This is the right time to think about your methods of study, and revise them as need be. Remember the Golden Rules:

1) Read each unit right through, from beginning to end, before you start studying in detail.
2) Complete *all* the practical problems. They have all been chosen to illustrate essential points of BASIC.
3) Don't go on to the next unit until you have mastered the present one. If you get really stuck, try going back a couple of units and repeating the work.
4) Don't rush matters. The later units in the course will take you four or five days each to absorb.

Good luck!

# UNIT:16

## THE DATA, READ AND RESTORE FUNCTIONS

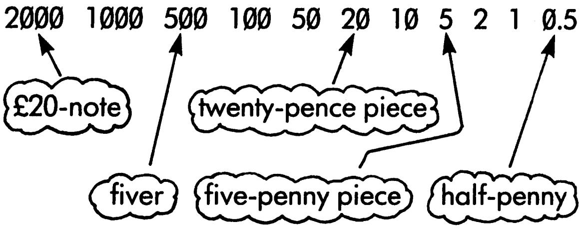
This unit introduces an important and useful group of commands which use the keywords DATA, READ and RESTORE.

Programs often need to refer to lists of words or phrases, or sets of numbers which don't follow any fixed arithmetical pattern. For instance, a program which calculates and displays the calendar for any selected year must be told the names of the days of the week, and of the months of the year. It also needs the sequence of days in each month, i.e.:

31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31.

## COIN ANALYSIS

Another common sequence is the set of values of British notes and coins. To avoid decimals, which can give rise to problems, we'll stick to pence:




People in charge of paying the wages in large organisations have to plan the operation carefully each week. One special problem lies in making sure that there is the right amount of change to make up each wage packet exactly, using as few notes and coins as possible.

There is a useful process called "coin analysis", which takes a sum of money and breaks it down into the smallest possible number of notes and coins. For example:

```
£158.37 = 7 AT 2000P (i.e. £20 notes)
          1 AT 1000P
          1 AT 500P
          3 AT 100P
          0 AT 50P
          1 AT 20P
          1 AT 10P
          1 AT 5P
          1 AT 2P
          0 AT 1P
          0 AT 0.5P (i.e. half-pence)
```

When this process is used on all the different wages to be paid, it helps the wages clerk to decide how many notes and coins of each sort to request from the bank.

We shall design a program for coin analysis. Maybe it is simple enough not to need a flow chart. Let's try.

We begin by asking the *user* to supply a figure for the wage to be broken down. The starting figure, in pence, is stored in W:

```
INPUT "WAGES IN PENCE";W
```

Next we extract the number of £20 notes needed and put this number into variable T. The right number is the result when W is divided by 2000, ignoring any remainder or fractional part. An appropriate command is

```
T = INT(W/2000)
```

Note that unless W is a sum of money which can be paid out exactly in £20 notes (like £80) W/2000 by itself will be a number with a decimal part, like 7.9185. The INT and brackets ensure that only the whole number part of this expression (such as 7) is used and the rest is discarded.

Next we display the number of £20 notes:

```
PRINT T; "AT 2000P"
```

We have now accounted for a substantial part of the wage packet, as much of it, in fact, as could be paid wholly in notes of £20 each. To continue the coin analysis process we must take away this amount from the starting wage, leaving only the remainder, which *must* be less than £20 or 2000 pence. The amount accounted for is 2000★T, so we put:

```
W = W − 2000 ★ T
```

(In the example, W would now be 15837 − 2000 ★ 7 = 1837.)

For the next stage we calculate and display the number of £10 notes and decrease W accordingly. We can use variable T again since its original value (the number of £20 notes) is no longer of any interest:

```
T = INT(W/1000)
PRINT T; "AT 1000P."
W = W − 1000★T
```

The following stages, right down to the final half-penny, are all very similar. We get:

```
10 INPUT "WAGES IN PENCE";W
20 T = INT(W/2000)
30 PRINT T; "AT 2000P."          £20 notes
40 W=W−2000★T
50 T = INT(W/1000)
60 PRINT T; "AT 1000P."          £10 notes
70 W = W − 1000★T
...
...
140 T = INT(W/50)
150 PRINT T; "AT 50P."           50-penny pieces
160 W = W − 50 ★T
...
...
320 T = INT(W/0.5)
330 PRINT T; "AT 0.5P."          half-pennies
340 W = W − 0.5★T
350 STOP
```

This program looks extremely repetitive. If we could somehow get the successive note and coin values into a variable — say V, then the whole 35-line program could be condensed into a single loop with the three commands:

```
T = INT(W/V)
PRINT T; "AT ";V;"P."
W = W – V★T
```

This loop would be executed 11 times; once for V=2000, once for V=1000, and so on down to V=0.5.

**USING A LOOP**

It would be pleasant and comfortable to use a FOR command, but this will not solve the problem because the values of V don't follow a set pattern. We must find another way of putting the values of the notes and coins in V, although we can still use a FOR to go round the loop 11 times.

Let's consider a 'silly' solution. We know that one way of getting numbers into a program is to have the user type them on the keyboard. We could always put an INPUT V command into the loop and compel the *user* to supply the sequence of numbers 2000 1000 500 100 ... 0.5. The program would read:

```
10 INPUT "WAGES IN PENCE";W
20 FOR J=1 TO 11
30 INPUT "NEXT VALUE";V
40 T = INT(W/V)
50 PRINT T; "AT ";V;"P."
60 W = W – T★V
70 NEXT J
80 STOP
```

Key this program in and try to run it. You will get a display something like:

```
WAGES IN PENCE? 9472
NEXT VALUE ? 2000
4 AT 2000 P.
NEXT VALUE ? 1000
1 AT 1000 P.
NEXT VALUE ? 500
0 AT 500 P.
...
```

**DATA AND READ STATEMENTS**

Of course there are many reasons why this is not practical.... It's very hard work for the user, and if a single mistake occurs in typing the sequence 2000, 1000, 500, 100... the whole coin analysis is ruined and has to be started again.

The designers of BASIC have overcome this problem in an ingenious and elegant way. Suppose the VIC could type its own numbers as it went along? It would use a 'phantom' keyboard so that the numbers didn't appear on the screen, and the typing would be done instantaneously, letting the machine run at its full speed. What about the numbers to be typed on the phantom keyboard? They would be entered in advance, in the form of DATA statements.

Now change line 30 of the program in your VIC and add some DATA lines to get:

```
  10 INPUT "WAGES IN PENCE";W
  20 FOR J=1 TO 11
  30 READ V                 This line changed
  40 T = INT(W/V)
  50 PRINT T; "AT ";V;"P."
  60 W = W – V★T
  70 NEXT J
  80 STOP
1000 DATA 2000          New lines from here on
1010 DATA 1000
1020 DATA 500
1030 DATA 100
1040 DATA 50
1050 DATA 20
1060 DATA 10
1070 DATA 5
1080 DATA 2
1090 DATA 1
1100 DATA 0.5
```

If you run this new version of the program, it will give you a coin analysis for any figure you supply. You will only have to input the figure to be analysed. The program has 8 lines of code, and then one DATA statement for each value of note or coin. This is clearly an improvement on the original version which had 35 commands.

Let's examine the program a little further. The READ command in line 30 is very similar to INPUT. The keyword READ can be followed by the names of one or more variables, which can be either numbers or strings. The important difference is that the command gets its information from a DATA statement embedded in the program rather than from the user. If you like, you can imagine a demon who lives inside the VIC, and who has his own private keyboard. Every time the VIC obeys a READ command the demon finds a DATA statement and rapidly types its contents on his keyboard. He remembers which statements he has used, and works down them in the order of their label numbers. Thus, when the machine executes the READ command at 30 for the first time, the demon finds the first DATA statement and types the number it contains — 2000. This value is allocated to V. The second time round, the value used is 1000, and so on.

Let's show how the READ command can handle strings as well as numbers. Suppose we want our coin analysis program to use descriptive names for the notes and coins, such as

```
1 FIVE-PENNY PIECE
```

instead of the cryptic

```
1 AT 5P.
```

The names we need can be included in the DATA statements alongside the values themselves. The READ command will now set up *two* variables: the value V and the corresponding name N$. The modified program with the changes in lines 30, 50 and the DATA statements is:

```
  10 INPUT "WAGES IN PENCE";W
  20 FOR J = 1 TO 11
  30 READ V,N$
  40 T = INT (W/V)
  50 PRINT T;N$
  60 W=W-V★T
  70 NEXT J
  80 STOP
1000 DATA 2000, TWENTY-POUND
     NOTES
1010 DATA 1000, TEN-POUND NOTE(S)
1020 DATA  500, FIVE-POUND NOTE(S)
1030 DATA  100, ONE-POUND NOTE(S)
1040 DATA   50, FIFTY-PENNY PIECE(S)
1050 DATA   20, TWENTY-PENNY PC(S)
1060 DATA   20, TEN-PENNY PIECE(S)
1070 DATA    5, FIVE-PENNY PIECE(S)
1080 DATA    2, TWOPENCE(S)
1090 DATA    1, PENNY(IES)
1100 DATA  0.5, HALFPENNY(IES)
```

# EXPERIMENT 16·1

Modify the coin analysis program so that it works for the monetary system of the United States of America. The values and their names are:

| | |
|---|---|
| $50 | FIFTY-DOLLAR BILL(S) |
| $10 | TEN-DOLLAR BILL(S) |
| $5 | FIVE-DOLLAR BILL(S) |
| $1 | DOLLAR(S) |
| $0.25 | QUARTER(S) |
| $0.10 | DIME(S) |
| $0.05 | NICKEL(S) |
| $0.01 | PENNY(IES) |

| *Experiment 16.1 Completed* | |
|---|---|

## FORMAT OF DATA STATEMENTS

There are a few simple rules you should know about DATA statements.

First, DATA statements may contain strings and numbers mixed in any order. The maximum length for one statement is 4 screen lines (88 characters). If a DATA statement includes more than one item, then the items are separated by commas. You don't need to put quote-marks round a string unless the string includes a comma or a screen control character such as SHIFT and CLR/HOME. For example, the DATA statement

```
DATA 21, QUEEN ST.
```

contains two items which are a number and a string, but

```
DATA "21,QUEEN ST."
```

contains only one: the string 21,QUEEN ST.

Second, the number of items in each DATA statement doesn't have to correspond with the number of variables in the READ command. Each READ takes just as many items as it needs, and if this uses up only part of a line it doesn't matter; the next READ will begin where the first left off. Likewise, a DATA statement which holds too few items will simply make the VIC go on to the next DATA statement as soon as it needs to.

To illustrate this point, the second coin analysis program will still work correctly if the data is rearranged as:

```
1ØØØ DATA 2ØØØ
1Ø1Ø DATA TWENTY-POUND NOTES
1Ø3Ø DATA 1ØØØ
1Ø4Ø DATA TEN-POUND NOTE(S)
```

or

```
1ØØØ DATA 2ØØØ, TWENTY-POUND
NOTES, 1ØØØ,TEN-POUND
NOTES,5ØØ,FIVE-POUND NOTES,1ØØ,...
```

or even

```
1ØØØ DATA 2ØØØ
1Ø1Ø DATA TWENTY-POUND NOTES,1ØØØ
1Ø2Ø DATA TEN-POUND NOTE(S),5ØØ,
     FIVE-POUND NOTE(S),1ØØ
1Ø3Ø DATA ONE-POUND NOTE(S)
...
```

In other words, the VIC demon rattles off all the contents of the DATA statements on his keyboard without being too worried about where one statement ends and the next one begins.

Third, the DATA statements are not part of the program in the same way as the various commands are. Whenever you type RUN, the DATA statements are effectively sorted out and put in a different pile *before* the first command is obeyed. This means that when a program is typed the DATA statements can be placed in front of, following, or in between the commands. For instance the coin analysis program could have been written with a DATA statement on every other line. However, this would be bad programming practice. Shuffling the program like this would make it difficult to read and is not recommended. A useful convention is to put all the DATA statements together either at the end, or at the beginning of the program, and to give them label numbers which are immediately recognisable.

## DATA AND READ STATEMENT ERRORS

Two types of error can happen with DATA statements and READ commands.

If you give a READ command when all the DATA statements have already been used up (or if there aren't any in the first place) you get an OUT OF DATA error. This explains what happens if you type RETURN when the cursor is on a line with READY. The VIC thinks you mean READ Y.

If the READ specifies a number variable, and the next item in the DATA statement isn't a number, you get a reported syntax error in the DATA statement (not the READ command). For instance if you put:

```
1Ø READ A
...
...
1ØØ DATA HELLO
```

you will get

```
? SYNTAX
ERROR IN 1ØØ
```

This can be confusing, since the *real* error is more likely to lie in the READ command; you may have meant to put:

```
1Ø READ A$
```

It is worth noting that there is no corresponding fault the 'other way round'; if you read a number into a string variable it will go in as a string of digits without reporting an error. Why shouldn't it? A string can be any sequence of characters, including a sequence of digits.

## THE RESTORE COMMAND

Finally we mention the command RESTORE. This command takes the VIC back to the beginning of the pile of DATA statements so that they can be read all over again. RESTORE can be used any time, even if the DATA statements haven't all been used up.

# EXPERIMENT 16•2

a) Write a program to display the months of the year, one per line. The *last* lines of your program should be

```
1ØØØ DATA JANUARY,FEBRUARY,MARCH,
     APRIL
1Ø1Ø DATA MAY,JUNE,JULY,AUGUST
1Ø3Ø DATA SEPTEMBER,OCTOBER,
     NOVEMBER,DECEMBER
```

b) Write a program to read a date (in the form day-month-year) and display the day and year in figures but the month in words. For example an input of:

22,6,1936

should give you 22 JUNE 1936

Use the same DATA statements as in the previous program. Write your program in the form of a loop including RESTORE so that when one date has been displayed another can be input.

| | |
|---|---|
| *Experiment 16.2 Completed* | |

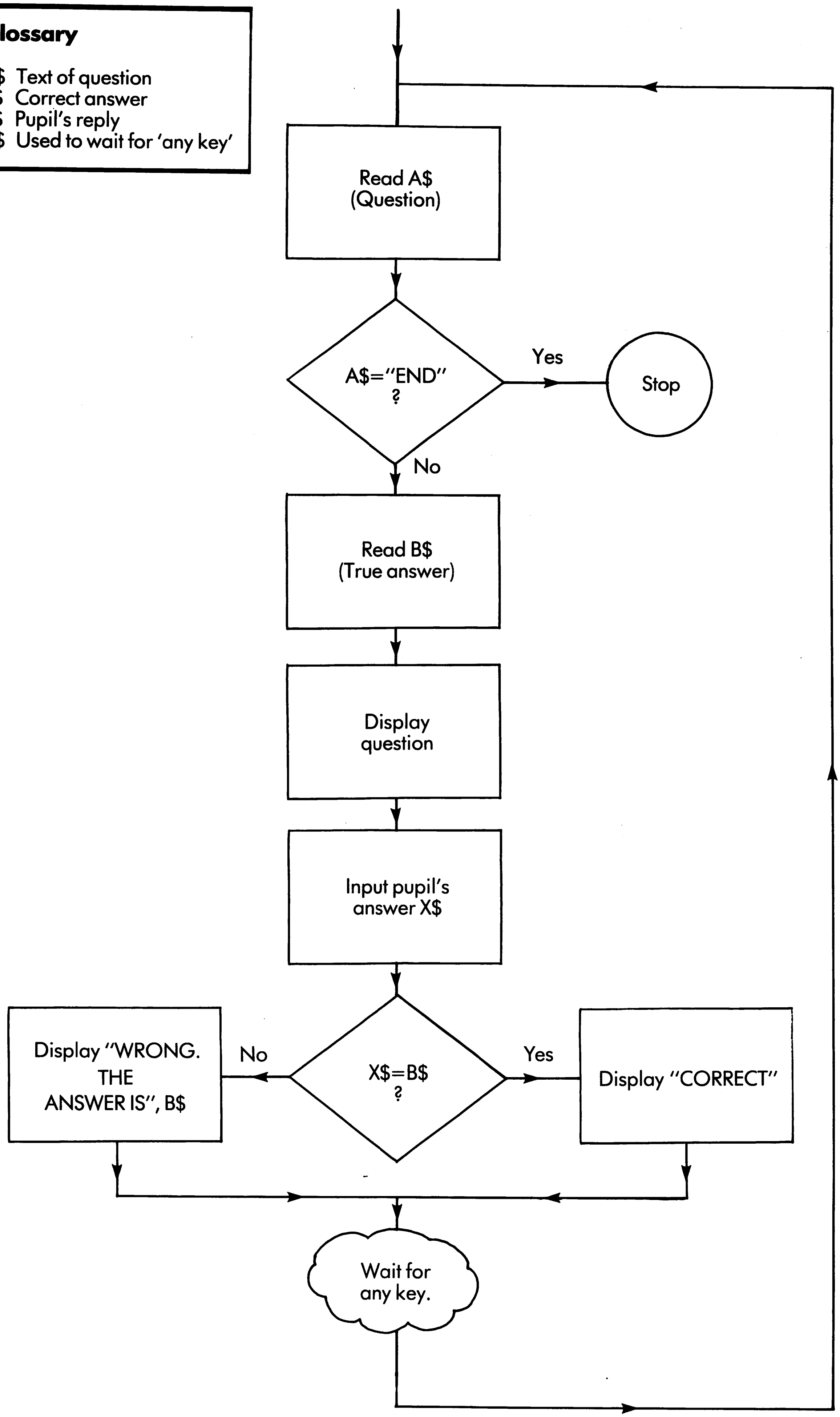
# EXPERIMENT 16•3

The DATA statement is invaluable in programs which present quizzes or questionnaires. Study the following program, enter it on your VIC and try it out:

```
10 READ A$
2Ø IF A$ = "END" THEN 19Ø
3Ø READ B$
4Ø PRINT " SHIFT and CLR HOME "
5Ø PRINT A$
6Ø PRINT
7Ø INPUT X$
8Ø PRINT
9ØIF X$ = B$ THEN 13Ø
1ØØ PRINT "WRONG. THE ANSWER IS"
11Ø PRINT B$
12Ø GOTO 14Ø
13Ø PRINT "CORRECT"
14Ø PRINT
15Ø PRINT "NOW PRESS ANY KEY"
16Ø GET C$
17Ø IF C$ = " " THEN 16Ø
18Ø GOTO 1Ø
19Ø STOP
5ØØ DATA WHAT IS THE CAPITAL OF
     FRANCE,PARIS
51Ø DATA WHAT COUNTRY HAS TOKYO
     AS ITS CAPITAL,JAPAN
52Ø DATA WHAT COUNTRY HAS THE
     LARGEST POPULATION,CHINA
53Ø DATA WHAT COUNTRY LIES DIRECTLY
     SOUTH OF THE USA,MEXICO
1ØØØ DATA END
```

**Glossary**

A$ Text of question
B$ Correct answer
X$ Pupil's reply
C$ Used to wait for 'any key'



When you have got the program running, make up some more questions and add them in, using label numbers 54Ø onwards.

This basic quiz program can be improved in various ways; for instance, you could give the pupil two or even more tries before displaying the right answer, and you could count the number of right answers and display a 'percentage' at the end of the lesson.

Write an improved quiz program to ask questions about your favourite subject.

| *Experiment 16.3 Completed* | |
|---|---|



The self test quiz for this unit is called UNIT16QUIZ.

# UNIT:17

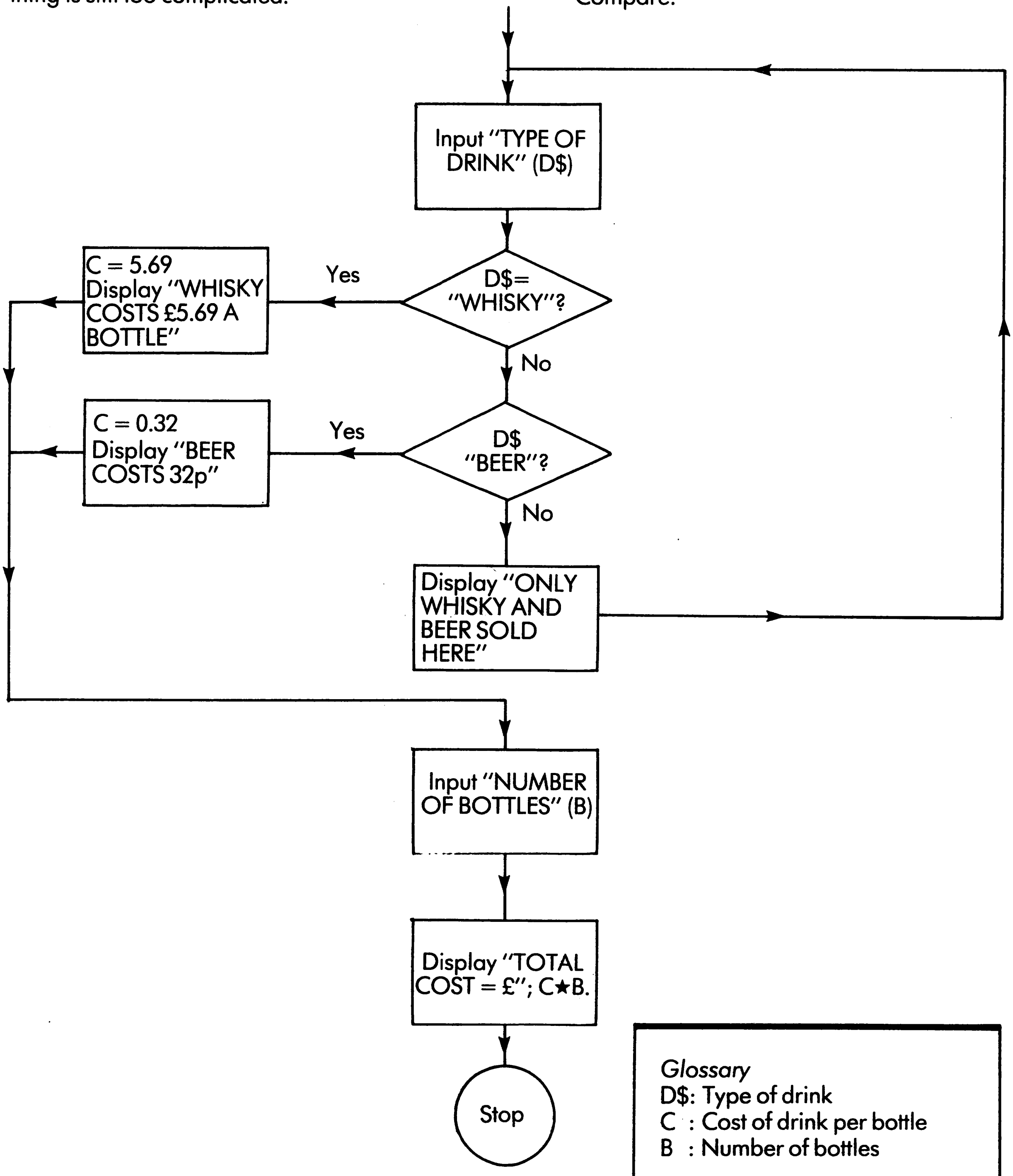
## DEALING WITH PROGRAM COMPLEXITY

By now you will have encountered the programmer's biggest problem — the control of complexity. Many people understand the principles of programming, and can write short, simple programs with ease; but when they apply the same methods to more complex problems, they have far less success. There seems to be a limit on the amount of detail we can hold in our heads at any time. When this limit is passed, the result is confusion, lots of crude errors, and programs which consistently give wrong answers. None of the aids described in Part 1 (such as tracing in Unit 8 and flowcharting in Unit 11) give very much help if used purely on their own. Everything is still too complicated.

In this unit we consider some of the ways of reducing the complexity of a program, without detracting from the job it is supposed to do. If your ambition is to be a serious programmer, you should study these methods and use them consistently in all your work. They will help you move forward towards solutions instead of being firmly stuck with a problem, trying random alterations to see if, by pure luck, you can hit on one that seems to work.

One of the main advantages of using flowcharts in program design is that they indicate the structure of a program much better than chunks of BASIC code.

Compare:




*Glossary*
D$: Type of drink
C : Cost of drink per bottle
B : Number of bottles

*Figure 17.1*

with:

```
10 INPUT "TYPE OF DRINK";D$
20 IF D$ = "WHISKY" THEN 70
30 IF D$ = "BEER" THEN 100
40 PRINT "ONLY WHISKY AND"
50 PRINT "BEER SOLD HERE"
60 GOTO 10
70 C=5.69
80 PRINT "WHISKY COSTS £5.69."
90 GOTO 120
100 C=0.32
110 PRINT "BEER COSTS 32P."
120 INPUT "HOW MANY BOTTLES";B
130 PRINT "TOTAL COST = £";C★B
140 STOP
```

The effect of translating the flowchart into BASIC is obvious; a clear set of instructions has been 'squashed' into a shapeless one-dimensional list of commands which must be disentangled before it makes any sense.

## USE OF THE COLON

Microsoft BASIC, the version of the language used on the VIC, has some useful features which allow much of the structure of a flowchart to be preserved.

A whole sequence of commands can be grouped together by putting them after the same label number, and separating them with the colon symbol ":" without a RETURN. The effect is the same as if the commands had been written out on separate lines, except that it is impossible to jump to any of the intermediate commands by a GOTO or an IF. For example the sequence

```
10 INPUT "WHAT IS YOUR NAME";N$
20 PRINT "HELLO ";N$
30 PRINT
40 S=0
50 Q=100
```

could be replaced by

```
10 INPUT"WHAT IS YOUR NAME";N$:
   PRINT"HELLO ";N$:PRINT:S=0:Q=100
```

provided that no other part of the program had a jump to any of the commands originally numbered 20 to 50. The limit to the number of commands which can be grouped together is set by the VIC's internal line length of 88 characters (4 screen lines).

## OTHER WAYS TO USE IF-THEN STATEMENTS

The THEN in an IF-THEN command need not always be followed by a label number, but can instead be succeeded by a command (or a group of commands) which is only executed if the condition is true. This means you could replace

```
10 IF X=0 THEN 30
20 GOTO 40
30 PRINT "X=0"
40 ———
```

by

```
10 IF X=0 THEN PRINT "X=0"
20 ———
```

A word of caution is due at this point. If the condition in an IF-THEN command is false, the VIC always transfers control to the next labelled statement. It follows that if an IF-THEN command is part of a group of commands separated by colons, then any commands which follow it will inevitably be skipped if the condition is false. This may not be what you intended! To illustrate the point, consider the sequence



```
10 IF X=0 THEN 20: Y=5: GOTO 30
20 Y=7
30 PRINT Y
```

Looking at the program, you can guess what was meant: the programmer wanted Y to be set to 7 if X was 0, or to 5 if it was not 0. Unfortunately this is not what actually happens. Consider the command on line 10:

```
IF X=0 THEN 20
```

If the condition is true (i.e., X=0) the VIC jumps to command 20, just as you would expect. If the condition is false, the machine follows the rule and transfers control to the next labelled statement, which just happens to be 20! The commands

```
Y=5 : GOTO 30
```

will never be obeyed under any circumstances.

This trap is avoided by following a simple rule: If an IF-THEN command involves a jump to a label, then it must be followed by a RETURN — not a colon.

Using these new facilities, the drink-price program we discussed earlier can be shortened and clarified:

```
10 INPUT "TYPE OF DRINK"; D$
20 IF D$="WHISKY" THEN C=5.69:PRINT
   "WHISKY COSTS £5.69.":GOTO 50
30 IF D$="BEER" THEN C=0.32:PRINT
   "BEER COSTS 32P.":GOTO 50
40 PRINT"ONLY WHISKY AND":PRINT
   "BEER SOLD HERE":GOTO 10
50 INPUT"HOW MANY BOTTLES";B
60 PRINT"TOTAL COST = £";C★B
70 STOP
```

Compare the two versions, and notice that the second one conforms much more closely to the structure of the original flow chart.

# EXPERIMENT 17·1

163

Rewrite the following programs, using as few labelled commands as you can:

a)
```
10 INPUT "HOW MANY MINUTES";M
20 R=TI+M★3600
30 IF TI<R THEN 30
40 PRINT "TIME UP"
50 STOP
```

b)
```
10 PRINT"USE 1000000 TO"
20 PRINT"END INPUT"
30 S=0
40 N=0
50 INPUT "NEXT NUMBER";X
60 IF X=1000000 THEN 100
70 S=S+X
80 N=N+1
90 GOTO 50
100 PRINT "AVERAGE = ";S/N
110 STOP
```

Load the program entitled UNIT17PROG and list it. You will see that it is supposed to recognise the names JIM, BOB, KATE and PENNY and tell you what they are short for. Unfortunately, the program doesn't actually work. Correct it.



| | |
|---|---|
| *Experiment 17.1 Completed* | |

## LOGICAL OPERATORS

Some further help in simplifying programs is given by the logical operators AND, OR and NOT. These words have special meanings in BASIC and are used to link up simple conditions in IF-THEN commands so that more complex decisions can be taken.

## THE "AND" OPERATOR

Let's begin with the most frequently used logical operator, AND. It generally comes between two conditions, like this:

```
IF A>18 AND M$ = "Q" THEN ...
```



The resulting compound condition (which is everything between the IF and the THEN) is only *true* if both the simple conditions are also *true*. If either (or both) is *false*, the compound condition is *false*.

The AND operator generally allows two or more IF-THEN commands to be replaced by only one. Consider a program which examines applications to join a security company. The rules say that recruits must be at least 18 years old and 64 inches tall. The flowchart is likely to include a section like:

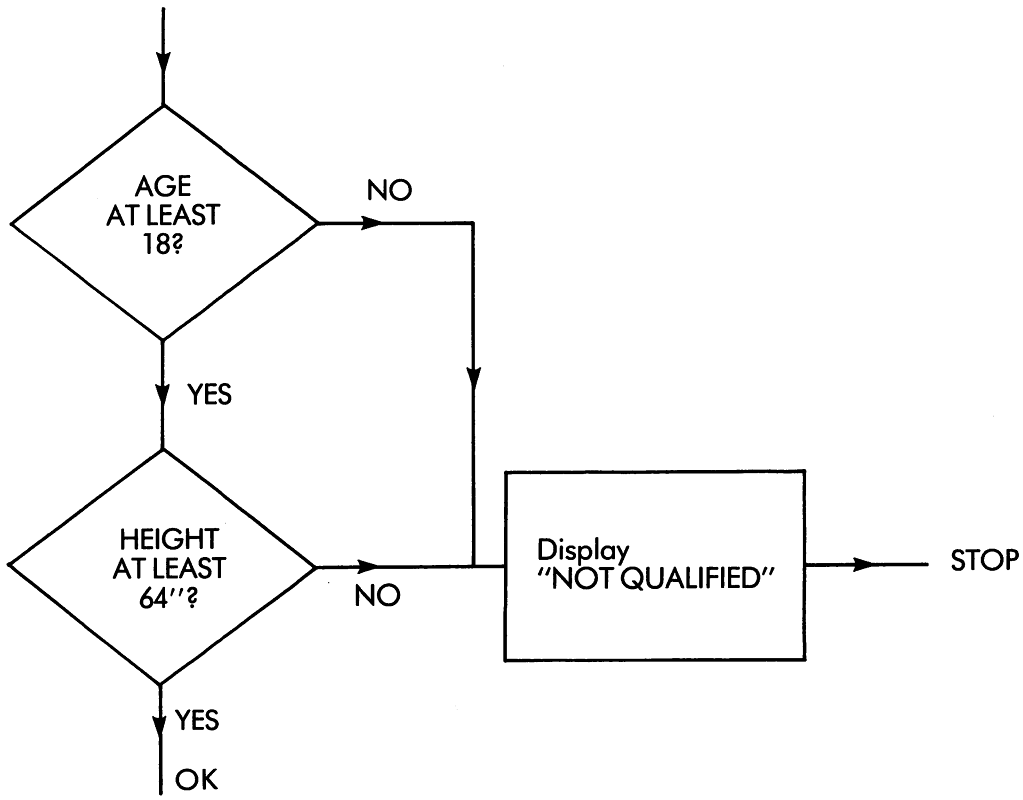



*Figure 17.2*

If only simple conditions were allowed, the sequence could be coded as:

```
100 IF A>=18 THEN 130
110 PRINT "NOT QUALIFIED"
120 STOP
130 IF H<64 THEN 110
140 REM AGE AND HEIGHT ACCEPTABLE
150 ...
```

Using a compound condition, which applies the tests for age and height at the same time, the section can be written much more compactly and with greater elegance. Its meaning is immediately clear:

```
100 IF A>= 18 AND H >= 64 THEN 120
110 PRINT"NOT QUALIFIED" : STOP
120 REM AGE AND HEIGHT ACCEPTABLE
```

AND is an operator, rather like ★, except that it uses conditions rather than numbers. This means that conditions can be chained together indefinitely, up to the limit of the internal line length of 88 characters. We could put:

```
IF A=5 AND B<7 AND X$<>"JOE"
AND ... THEN 1260
```

The resulting compound statement is only true if *all* the simple conditions are also true.

One special use of AND is to decide whether a variable lies within a specific range. We may want to test if the value of a variable X lies between, for example, 7 and 12 inclusive. A mathematician would express this idea by writing

$$7 <= X <= 12$$

but we couldn't put such a "condition" into an IF-THEN command — it isn't BASIC. Instead, we use two simple conditions linked with an AND:

```
IF X>=7 AND X<=12 THEN ...
```

## THE "OR" OPERATOR

The OR operator is used in much the same way as AND, but it produces a true condition if *either* or *both* of the two constituent conditions are true. One common use of OR is to check that the reply to a question is one of those intended. For example:

```
10 PRINT "ARE YOU BALD? ANSWER"
20 INPUT "YES OR NO";A$
30 IF A$ ="YES" OR A$ ="NO" THEN 50
40 PRINT "ANSWER THE QUESTION!":
   GOTO 10
50 REM LEGAL ANSWER RECEIVED
```

The compound condition can be expanded to include several OR's, as in

```
IF H$="BLACK" OR H$="BROWN" OR
H$="RED" OR H$="FAIR" THEN 30
```

Notice that one form you can't use (since it is not correct BASIC) is

```
IF H$ = "BLACK" OR "BROWN" OR "RED"
OR "FAIR" THEN 30
```

Not correct BASIC

## COMBINATIONS OF LOGICAL OPERATORS



It is often convenient to use compound conditions which use more than one kind of logical operator. For example, the rules for issuing driving licences may say that the applicant must be over 16 to drive a motorcycle, over 17 to drive a car and over 21 to be in charge of a bus. We can put

```
IF V$="MOTORCYCLE" AND A>=16
OR V$="CAR" AND A>=17
OR V$="BUS" AND A>=21
THEN PRINT "OK"
```

When the VIC comes to work out this compound condition, it does so in a particular order; first the simple conditions themselves, then the AND's, and lastly the OR's. In this example, the process gives exactly the result we need.

In practice, compound conditions may not always be so easy to write. Consider the example of a firm looking for a programmer with three years' experience of either the BASIC or the COBOL programming language. If we put

```
IF E>=3 AND L$="BASIC" OR
         L$="COBOL"
```

(E is the number of years experience, L$ the language)

the rules of evaluation will select either:

a) A BASIC programmer with at least 3 years' experience

or

b) A COBOL programmer with possibly no experience at all.

This is because AND is used first, and associates E>=3 with "BASIC" but not with "COBOL".

The way to avoid this problem is to use brackets. Just as in ordinary algebra, everything inside brackets is worked out before anything outside. By writing

```
IF E>=3 AND (L$="BASIC" OR L$=
"COBOL")
```

we get the correct order and therefore the right meaning.

**THE "NOT" COMMAND**

NOT is the third logical operator. It is applied to a condition (simple or compound) and reverses its sense. For example, if X>5 is true, then NOT X>5 is false, and vice versa.

It is never necessary to use NOT with simple conditions, since the 'opposite' relation can be used instead. Thus

NOT X=5 is the same as X<>5
NOT X<>5 is the same as X=5
NOT X<5 is the same as X >=5
NOT X>5 is the same as X<=5
NOT X<=5 is the same as X>5
NOT X>=5 is the same as X<5

NOT comes into its own with compound conditions, as we shall see below.

The rules of BASIC specify that in the absence of brackets, NOT is to be used before any other logical operator. We have seen that it isn't sensible to make it invert simple conditions. To make it turn round the whole of a compound condition, the condition must be enclosed in brackets, like this:

```
IF NOT (X=5 AND Y=7) THEN . . .
```

A compound condition with a NOT could be used to detect and refuse certain banned replies. This is illustrated by the sequence:

```
10 PRINT "WHAT DO YOU THINK"
20 INPUT "OF THAT";R$
30 IF NOT (R$="BLAST" OR R$=
   "BOTHER" OR R$="CURSES") THEN 50
40 PRINT "MIND YOUR LANGUAGE!":
   GOTO 10
50 REM REPLY ISN'T RUDE
```

Compound conditions with NOT in front of them can often be simplified. If every logical operator inside the brackets is an OR, then the NOT and the brackets can be taken away provided that

a) Each simple condition is inverted
b) Every OR is changed into an AND.

Thus line 30 in our example could be written

```
30 IF R$<>"BLAST" AND R$<>"BOTHER"
   AND R$<>"CURSES" THEN 50
```

A similar rule applies to inverted compound conditions in which every operator is an AND: the AND's become OR's, the simple conditions are inverted and the NOT and the brackets taken away. These two rules are called "De Morgan's Laws" after their discoverer. Most programming text books recommend that NOT conditions be avoided wherever possible and it is usually easier to do so.

# EXPERIMENT 17·2

a) The Customs regulations say that you can import without paying duty:
"One litre of spirits and two litres of fortified wine or four litres of still wine."

S, F and W are variables which denote the quantities of spirits, fortified wine and still wine, respectively. The regulation itself is ambiguous; so write down two compound conditions, each of which is intended to be true if duty is payable. The first of your answers should interpret the rule in the most generous sense to the traveller; the second, in the most restrictive sense.

b) Use De Morgan's Laws to express the following compound conditions without using NOT's:

NOT (N$="JONES" OR N$ ="SMITH"
OR N$= "BROWN")
NOT (X<= 15 AND X >= 4)

c) Suppose that a program has measured a reaction time (in seconds) and placed it in variable T. Write a section of code which will display one of the following comments, as may be appropriate:

| *Value of T* | *Comment* |
|---|---|
| T < 0.1 | FANTASTIC!! |
| 0.1 <= T < 0.15 | AMAZINGLY GOOD! |
| 0.15 <= T < 0.2 | VERY GOOD |
| 0.2 <= T < 0.25 | GOOD |
| 0.25 <= T < 0.28 | FAIR |
| 0.28 <= T < 0.33 | PRETTY SLOW |
| 0.33 <= T < 0.4 | WAKE UP! |
| 0.4 < T | TRY AGAIN WHEN YOU'RE SOBER!! |

d) My encyclopaedia is in four volumes:

| | | |
|---|---|---|
| 1: | ABRAHAM | to FRANCE |
| 2: | FRANCHISE | to LEVANT |
| 3: | LEVITATION | to QUOIT |
| 4: | QUOTIENT | to ZYLOPHONE |

Write a program which inputs any word and tells me in which volume to search for it. The program should also tell me if the word is not included (e.g., "QUORUM").

Hint: Remember that the operators <, >= and the others like them can be used with strings, and give results according to alphabetical order.



| | |
|---|---|
| *Experiment 17.2 Completed* | |

# UNIT:18

171

## INTRODUCING SUBROUTINES

Up to this point in the course, you have gained plenty of practice in writing small programs — say up to 3Ø commands or so. Sooner or later, you are bound to discover that most interesting programs have to be very much longer, and that you need specialised tools and techniques to help you build them correctly.

A vital aid in writing large programs is the subroutine facility. In flow-charting (Unit 11 of Part 1) you have already come across the idea that you can put complicated actions inside 'clouds' and leave the coding till after. This turns out to be a useful method of dealing with complexity, because it allows you to forget lots of details and concentrate on the main issues of the problem.

Subroutines are quite like clouds. A subroutine consists of a group of commands which co-operate to do some particular and well-defined task. When you design a program, you can think of this task as a single step, no matter how complicated the task may actually be.

We'll start with a very simple subroutine. Consider a program that quizzes you on sums.

### Glossary

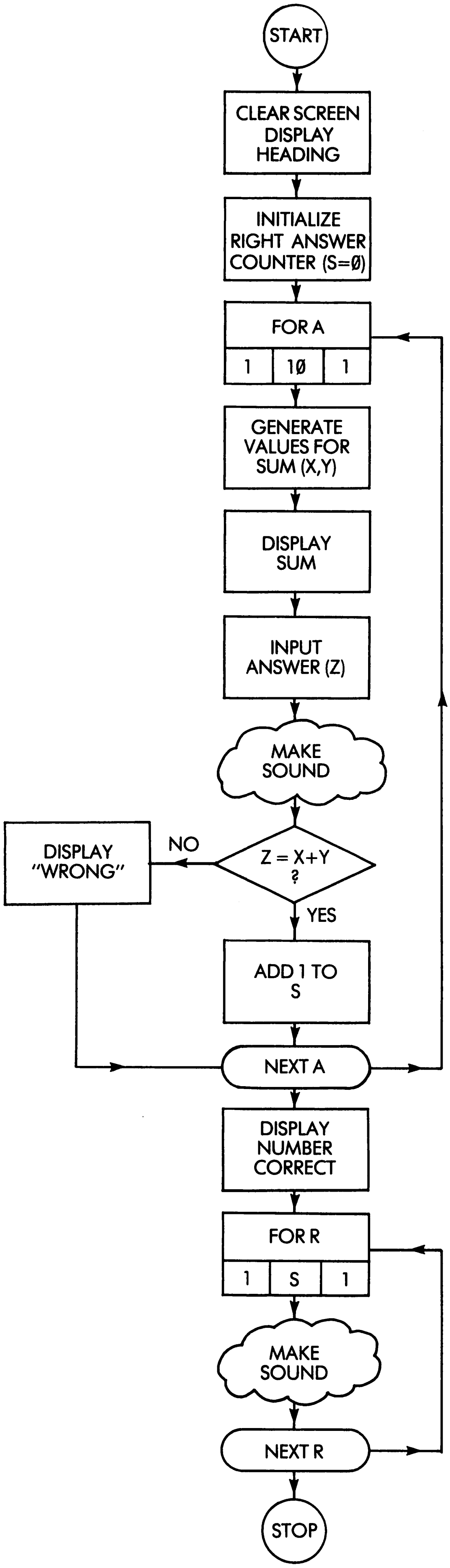
S: Counter of number of correct answers

X, Y: Values to be added together

Z: Sum (answer)

A: Counter of number of questions

R: Counter of reward loop



The task we select for making into a subroutine is the cloud labelled

"Make Sound"

The code for this action would normally be something like:

```
1Ø POKE 36878,15;POKE 36876,245
2Ø FOR M=1 TO 1ØØ:NEXT M
3Ø POKE 36878,Ø
4Ø FOR M=1 TO 8ØØ:NEXT M
```

To turn these instructions into a subroutine they have to be "clothed" so that they fit properly into the main program. In technical language, we must provide an *interface*.

First, we choose a set of label numbers so high that they do not clash either with the main program or with any other subroutine. The highest label number allowed is 63999.

Second, we add a descriptive REMark at the beginning of the subroutine. This is not absolutely necessary, but is a mark of competent programing and is very handy since many subroutines can be used in other programs.

Third, we assign a distinctive name to the variable used by the subroutine such as a double letter. Again this is not compulsory but follows a useful convention we'll explain later.

Finally, we follow the code with the special command:

RETURN

This command, which is used at the end of every subroutine, must be spelled out in full (6 letters) and followed, as usual, by the RETURN key. Don't confuse the RETURN command and the RETURN key — they are totally different. Using these conventions the instructions for the subroutine could be:

```
1ØØØ REM SUBROUTINE TO MAKE PIP
     SOUND
1Ø1Ø POKE 36878,15:POKE 36876,245
1Ø2Ø FOR MM= 1 TO 1ØØ: NEXT MM
1Ø3Ø POKE 36878,Ø
1Ø4Ø FOR MM=1 TO 8ØØ:NEXT MM
1Ø5Ø RETURN
```

Now the main program can be written. Whenever there is a cloud with the instruction "MAKE SOUND" it can be translated by the subroutine call command:

GOSUB 1ØØØ

usually followed by a REM on the same line to explain what is being done. The whole program (including the subroutine) turns out like this:

```
  1Ø PRINT " SHIFT and CLR HOME
     ARITHMETIC TEST"
  2Ø PRINT "ANSWER THE FOLLOWING
     SUMS"
  3Ø S=Ø
  4Ø FOR A=1 TO 1Ø
  5Ø X=INT(1Ø★RND(Ø)+1)
  6Ø Y=INT(1Ø★RND(Ø)+1)
  7Ø PRINT X;"+";Y;"=";
  8Ø INPUT Z
  9Ø GOSUB 1ØØØ: REM MAKE SOUND
 1ØØ IF Z=X+Y THEN 13Ø
 11Ø PRINT "WRONG"
 12Ø GOTO 15Ø
 13Ø PRINT "RIGHT"
 14Ø S=S+1
 15Ø NEXT A
 155 FOR T =1 TO 5ØØ:NEXT T
 16Ø PRINT "THAT'S";S;"RIGHT OUT OF
     1Ø"
 17Ø FOR R=1 TO S
 18Ø GOSUB 1ØØØ: REM MAKE SOUND
 19Ø NEXT R
 2ØØ STOP
1ØØØ REM SUBROUTINE TO MAKE PIP
     SOUND
1Ø1Ø POKE 36878,15:POKE 36876,245
1Ø2Ø FOR MM=1 TO 1ØØ: NEXT MM
1Ø3Ø POKE 36878,Ø
1Ø4Ø FOR MM=1 TO 8ØØ: NEXT MM
1Ø5Ø RETURN
```

Key this program in, and verify that it works as you would expect.

## HOW GOSUB WORKS

Let's follow the program through. The GOSUB is very like a GOTO, but with one important difference. When the VIC jumps to the subroutine, it remembers the label of the next command in the main program, and stores this information in a special part of the memory called the stack. The RETURN at the end of the subroutine is also like a GOTO, but the destination is always the label number stored in the stack! This provides an automatic mechanism which ensures that whenever the VIC finishes executing a subroutine, it always gets back to the right place in the main program. The label number stored in the stack while the VIC is obeying the subroutine is called a *link* or *return address*.

Our program begins by obeying the commands in line 1Ø in the ordinary way. Command 9Ø says GOSUB 1ØØØ; so the computer jumps to 1ØØØ, but on the way it notes the label of the the command following GOSUB (which happens to be 1ØØ) and puts "1ØØ" into the stack. In this case "1ØØ" is the link.

Once it reaches the subroutine, the machine executes the commands 1Ø1Ø-1Ø5Ø the last line of which is RETURN. Since the stack contains "1ØØ", the RETURN is equivalent to a "GOTO 1ØØ", and so the machine returns to the main program at the right place.

When all ten sums have been answered the program comes to another subroutine call. It again jumps to line 1000, but this time the link placed in the stack is "190" instead of "100". When RETURN is obeyed the second time, control returns to line 190, not 100 as previously. The subroutine here is part of a loop. The control variable for that loop is S, the number right, so the subroutine will be called once for each sum answered correctly.

This simple example shows how you can split off one of the jobs which make up a program and treat it on its own. Clearly, the more complex the function you separate off, the more you simplify the overall design of the program. The example also shows how you can 'call' a subroutine from more than one place without writing it out each time. This may be true, but beware of people who tell you that the main value of subroutines is to shorten programs. This is false and misleading. The real point of subroutines is to simplify program structure by separating off complex sections and allowing them to be considered in isolation. This will be more obvious in subsequent illustrations.



# EXPERIMENT 18·1

a) Modify the program on page 172 so that the border turns to black when an answer is wrong and to purple when it is correct. Don't forget to restore the initial colour when the next sum is displayed.

b) Now use the same subroutines to write a completely different program. Imagine a very young child being taught to count. For each question, the program gives out a series of pips (between 1 and 9). The pupil is expected to count the pips and type the right numbers. For instance, ... pip — pip — pip ... has the right answer "3", and anything else is wrong.

| | |
|---|---|
| *Experiment 18.1 Completed* | |

## SUBROUTINES WITH VARIABLE TASKS

Experiment 18.1 shows you that subroutines can be quite independent of the program they live in. They can be moved from program to program, and they can be written by different people. (*You*, the reader, have just used *my* subroutines in your latest program.) You can actually buy libraries of subroutines for doing various tasks, and this can save a great deal of time when building a program.

A program with subroutines is a bit like an office with a boss (the main program) and several personal assistants (the subroutines). Each assistant specializes in doing just one job, such as fetching the top document from a filing cabinet or making coffee. The boss has a telephone, and can call an assistant at any time and tell him to do his special job. Then (at least in BASIC) the boss waits until the assistant rings back and says, 'ready'.

The subroutines we have already examined were severely limited. Each of them could only do one quite precise job, such as changing the border colour or making a particular kind of noise. In an office where the assistants are equally inflexible in what they can do, the only command the boss ever need give is "Do it!". That starts the coffee assistant making *one* cup of coffee, which is the only thing he understands how to do. If the boss has visitors and wants five cups, he has to call the coffee maker five times.

Assistants in this office would be much more useful if the boss could in some way qualify the job he gives them. For instance, it would save time if the coffee maker were able to count, and the boss could tell him *how many* cups to make. It would also be helpful if the boss could tell the archivist *which* document to fetch from the filing cabinet. The assistants would still be limited to one job, but they could do it in a more flexible way.

In the same general way, a subroutine in a program becomes much more useful if it can be asked to do any one of a whole family of related tasks. For instance, it might be convenient for a program to use a subroutine which gives out any number of 'bleeps' according to instructions from the main program.

This idea raises the interesting question of communication. You'd expect the messages which pass between the boss and his new, versatile assistants to be more complicated, since he now has to indicate a number or a document title. Assistants who have the special job of finding out information for the boss can pass this information back to him when they say 'ready'. In the office all this is quite simple because there is a telephone system, but what happens in programs?

## PASSING PARAMETERS TO AND FROM SUBROUTINES

Many programming languages, such as PASCAL or ADA, have special mechanisms for communicating between the main program and the subroutines, but BASIC does things in a much simpler way. The information is passed in *variables* which are shared between the main program and its subroutines. These variables have a special name: *parameters*. Any variable will do as a parameter, but we shall adopt a special convention: every parameter name shall consist of a letter followed by digit 1, followed by the $ sign if the parameter is a string. Examples are:

A1 X1 C1$ G1

Here is an example of a subroutine to display a line with any number of ★'s, like:

★★★

or

★★★★★★★★★★★★

```
3ØØØ REM DISPLAY NUMBER OF ★'S GIVEN
        IN X1 ON ONE LINE
3Ø1Ø FOR JJ=1 TO X1
3Ø2Ø PRINT "★";
3Ø3Ø NEXT JJ
3Ø4Ø PRINT
3Ø5Ø RETURN
```

Let's examine this routine closely. Commands 3Ø1Ø to 3Ø3Ø form a loop which is obeyed X1 times. Each time round, a ★ is displayed on the same line as the previous ★'s. X1 is the *parameter*, or variable which tells the subroutine how many ★'s are needed. JJ is a *local variable*; that is, it is used inside the subroutine, but its value outside of it is of no interest.

The subroutine is of no use unless it is called. This requires a pair of commands: one to set X1 to an appropriate value, and one to do the actual calling. To get a line of 17 ★'s, your program could include:

```
X1=17
GOSUB 3ØØØ
```

The value of a parameter can be set in several ways, such as by READ, INPUT or FOR commands as well as by simple assignment. To display a lop-sided pyramid we would write:

```
1Ø FOR X1=1 TO 18
2Ø GOSUB 3ØØØ
3Ø NEXT X1
4Ø STOP
```
(followed by the ★-displaying subroutine itself).

Key in this program (with the subroutine) and check that it works as you would expect.

## DECISIONS WHEN DESIGNING SUBROUTINES

Let's examine some of the decisions made while this program was being written. During the initial design, the programmer discovered that he needed a subroutine to display a variable number of ★'s on one line. He then decided, for no special reason, to put the subroutine at 3ØØØ and to use X1 as the parameter. At this stage, he could equally well have put the subroutine anywhere else (say 45ØØ) and he could have chosen a different variable (such as N1) to be the parameter.

Once the decision was made, however, matters were much more restricted. The subroutine now had to start

```
3ØØØ REM .....
```

Calls had to use X1 as parameter and be written as



```
GOSUB 3ØØØ
```

This is a good illustration of a general point: when you start designing a program you have plenty of freedom to do things in different ways; but as you make one decision after another your freedom gets less and less until at the end there is only one way left to go.

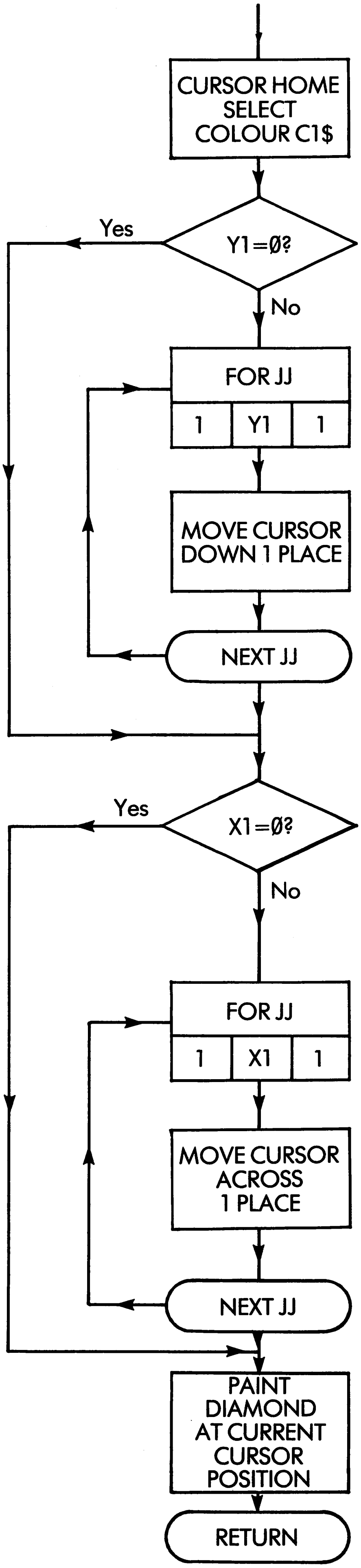
## USING MORE THAN ONE PARAMETER

Subroutines are not limited to one parameter only, but can use any (reasonable) number of them. Here, for instance, is a subroutine which displays a coloured diamond at any position on the screen. The parameters are:

X1 : Number of spaces across the screen
Y1 : Number of lines down the screen
C1$: Colour of diamond

**Glossary**

C1$: Colour of diamond
X1, Y1: Position of diamond
JJ: Used to count cursor movements



The corresponding code is:

```
2ØØØ REM DISPLAY C1$-COLOURED
     DIAMOND AT X1 SPACES ACROSS
     SCREEN AND Y1 PLACES DOWN
2Ø1Ø PRINT " CLR HOME ";C1$;
2Ø2Ø IF Y1=Ø THEN 2Ø6Ø
2Ø3Ø FOR JJ=1 TO Y1
2Ø4Ø PRINT " CRSR ";
2Ø5Ø NEXT JJ
2Ø6Ø IF X1 = Ø THEN 21ØØ
2Ø7Ø FOR JJ=1 TO X1
2Ø8Ø PRINT " CRSR ";
2Ø9Ø NEXT JJ
21ØØ PRINT " CTRL and RVS ON
CTRL and RVS OFF CRSR
SHIFT and CRSR SHIFT and CRSR
";
211Ø RETURN
```

This subroutine uses cursor commands in strings to move the cursor round the screen. 21ØØ paints a diamond. It looks frightening when written out in full, but the string includes only nine characters: Reverse on, reverse off, and (twice each) and three cursor movements to get from the end of the first row of graphics to the beginning of the second. The characters in the string are exactly those you would use if you were drawing a diamond directly from the keyboard.

You will remember that in VIC BASIC, FOR commands go round the loop at least once even if the final value is less than the starting value. The test for Y1=Ø is included so that the FOR loop in lines 2Ø3Ø-2Ø5Ø can be skipped altogether if necessary. The test for X1=Ø is there for a similar reason.

To test the subroutine, here is a program which fills the screen with green diamonds:

```
1Ø PRINT " SHIFT and CLR HOME ";
2Ø C1$= " CTRL and GRN "
3Ø FOR X1=1 TO 19 STEP 3
4Ø FOR Y1=Ø TO 21 STEP 4
5Ø GOSUB 2ØØØ: REM DRAW C1$-
   COLOURED DIAMOND AT X1,Y1
6Ø NEXT Y1
7Ø NEXT X1
8Ø GOTO 8Ø: REM LOOP STOP
```

# EXPERIMENT 18·2

Write a subroutine, starting at line 5ØØ, which draws a 'monster' in colour C1$, 2 lines below the top of the screen and X1 spaces from the left. The monster can be as simple or as complicated as you like.

Now add your subroutine to the following program, which will make your monster move across the screen from left to right:

```
1Ø PRINT " SHIFT and CLR HOME ";
2Ø FOR X1 = Ø TO 16
3Ø C1$= " CTRL and RED "
4Ø GOSUB 5ØØ: REM DRAW RED MONSTER
   AT X1
5Ø FOR T=1 TO 15Ø: NEXT T: REM WAIT A
   BIT
6Ø C1$= " CTRL and WHT "
7Ø GOSUB 5ØØ: REM ERASE MONSTER BY
   DRAWING IT IN WHITE
8Ø NEXT X1
9Ø GOTO 9Ø: REM LOOP STOP
```

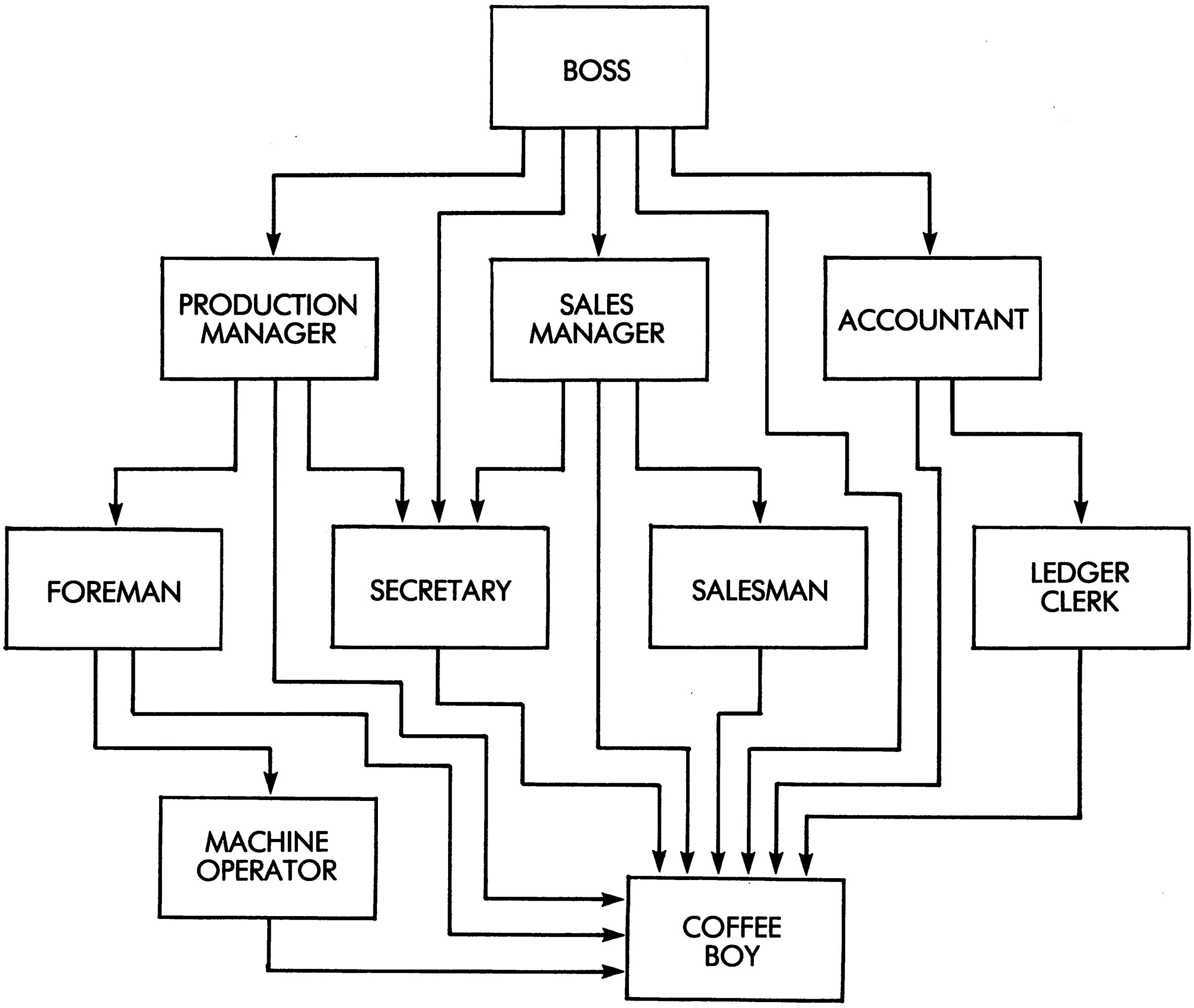
| *Experiment 18.2 Completed* | |
|---|---|

## MORE COMPLEX SUBROUTINES

Most offices aren't as simple as the one we described earlier in this unit. Generally speaking, the boss is helped by several 'executives', each of whom may have one or more personal assistants. These assistants may in turn have their own helpers, and so on down the chain of command. Some people can do their particular job for almost anybody else in the office; for example, the coffee boy has to serve anyone. The boss may ask for coffee for himself or may ask his secretary to ask for him.

To clarify the command structure, the office may have a chart somewhat like this:



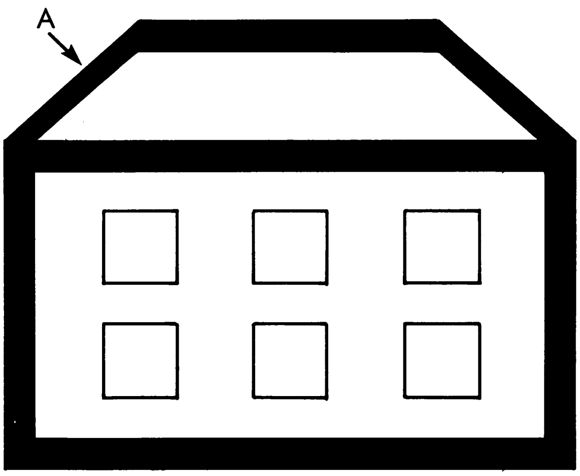



This type of structure can be reflected in a BASIC program. A subroutine may be called by the main program or it may be called by other subroutines, to any 'level'. The computer doesn't get mixed up because it makes special arrangements to store all the links it needs to get back to the right place in the main program. The stack where the links are stored is not just a single memory call, but has a separate position for each 'level' of call.

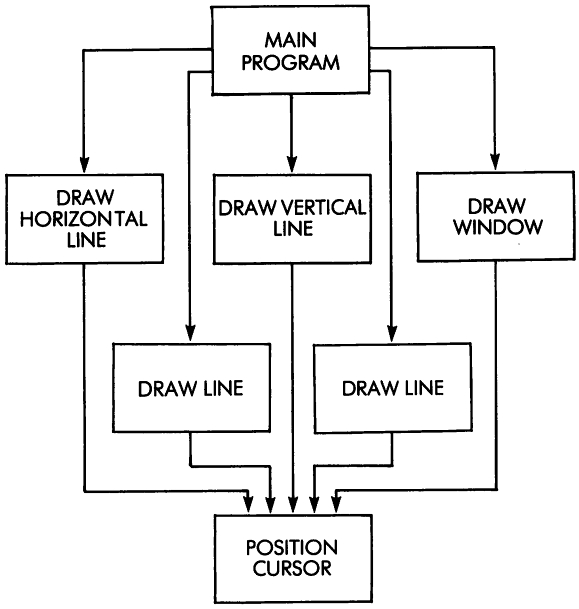
# EXPERIMENT 18·3

Load and run the program called PICTURE from the cassette tape. It generates a crude but recognisable drawing of a house. The drawing is made by calling a set of subroutines, one for each line or window in the picture. Thus the subroutine at line 1ØØØ draws a horizontal line from left to right. The line is N1 units long and starts at the location X1 spaces across the screen and Y1 lines down. Similarly, the subroutine at 2ØØØ draws lines downwards, the subroutine at 3ØØØ diagonally upwards from left to right, and the one at 4ØØØ, diagonally downwards from left to right. Thus to draw the line marked A in the picture, the calling sequence is

X1=1: Y1=8: N1=5: GOSUB 3ØØØ




List and examine the code in the various subroutines. They all begin with a common task: placing the cursor into the position indicated by X1 and Y1. Since this is a well-defined job, it is sensible to turn it into a subroutine on its own account. The 'power structure' of our program is therefore






Now erase commands 1Ø to 14Ø, and use the subroutines to draw a picture of your own choice. It could be a castle or a factory with a chimney. You could define a new subroutine to draw arched windows and draw a church.

| *Experiment 18.3 Completed* | |
|---|---|

Subroutines are an important topic, and we shall continue to discuss them in the next unit.

# UNIT: 19

181

## MORE ABOUT SUBROUTINES

In this unit we continue the study of subroutines. The key to writing robust useful programs and getting them to work quickly is a collection of techniques called 'software engineering'. These techniques aren't usually mentioned in introductory books, because they are generally considered to be professionals' tools. There seems no point, however, in learning to program badly when it is just as easy to do it well straight away.

## SUBROUTINE SPECIFICATION

One vital idea in software engineering is the *subroutine specification.* This is an exact description of *what* a subroutine does, and how (i.e. through which parameters) it communicates with the main program. The subroutine specification says *nothing* about the internal mechanism of the subroutine itself, or how it achieves the task it is set to do.

Subroutine specifications serve two quite different purposes. First, they can be printed in a catalogue of subroutines, so that other programmers can select useful subroutines for their programs and make all the practical arrangements for calling them. Second, a subroutine specification gives a firm starting point for the programmer who writes the subroutine itself. He can write it any way he likes so long as it does exactly what the specification says. Notice the order of things: specification *before* program. The only exception is that certain items may have to be added to the specification after the subroutine has been written.

Let's move straight to an example. Suppose you are writing a program to help young children do sums with fractions, like "1/6 + 1/3", or "5/8 × 2/3". If the sums are generated at random, then somewhere the program will have to work out its own answers to the questions it asks. You'll remember that when you are working with fractions, you're liable to come up with answers which aren't in their lowest terms. For instance you could get:

$$\text{"}1/6 + 1/3 = \frac{1+2}{6} = 3/6\text{" or "}5/8 \times 2/3 = 10/24\text{"}$$

The fractions "3/6" and "10/24" are not wrong, but they must be simplified before they can be accepted as totally correct.

The job of simplifying fractions is a self-contained task, clearly fitted for making into a subroutine. This subroutine will be different from the ones in Unit 18 in one important way: it will take in parameters from the main program, do a calculation and return results *to the main program.* It won't display anything on the screen (except possibly in an emergency) or require any input from the user. As far as the subroutine is concerned, the external world *is* the main program!

We begin by identifying and naming the parameters. To do its job the subroutine needs a *pair* of numbers (like 3 and 6 or 1Ø and 24) which are the 'top' and 'bottom' of the fraction it is trying to simplify. We'll choose A1 and B1 to represent the original values. A1 and B1 are called *input parameters,* even though the input is from the main program, and not (at least directly) from the user.

The result of the subroutine is another pair of numbers like 1 and 2 or 5 and 12. We'll make the subroutine return these values in C1 and D1. As you'd expect, C1 and D1 are called *output* parameters, even though there is no PRINTing to be done by the subroutine.

Finally we'll decide at random to put the first command of the subroutine at 55ØØ. This number doesn't clash with any other subroutine in our collection.

We can now write down a formal specification:

**Provisional Subroutine Specification**

*Purpose:* To simplify fractions to their lowest terms

*Line numbers:* 55ØØ to $\boxed{\quad\quad}$

*Parameters:* Input A1 (top of fraction)
B1 (bottom of fraction)
Output C1 (top of simplified fraction)
D1 (bottom of simplified fraction)

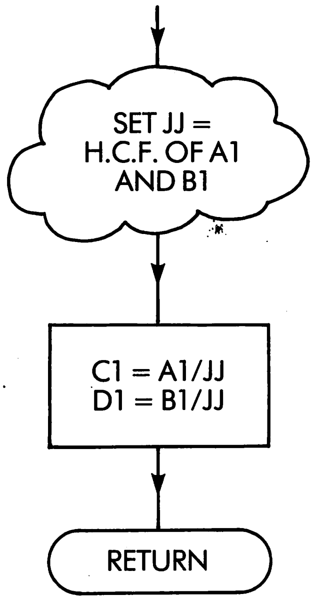
*Local Variables:* $\boxed{\quad\quad}$

Note that the specification is *provisional* and contains two empty boxes. One is intended for the last line of the subroutine, and the other is meant for a list of the variables used by the subroutine itself. These boxes can't be filled in until the subroutine has been written.

## SUBROUTINE TO SIMPLIFY FRACTIONS

Now we turn to the subroutine itself. To simplify a fraction, you first find the Highest Common Factor ("HCF") of the two numbers (some people call it the "GCD" or "Greatest Common Divisor") and then divide it into both of them. For example the HCF of 1Ø and 24 is 2, and 1Ø/24 in its lowest terms is therefore 5/12.

This process is easily flow-charted. We don't yet know how to work out the HCF of two numbers, so we'll use a cloud:




**Glossary**

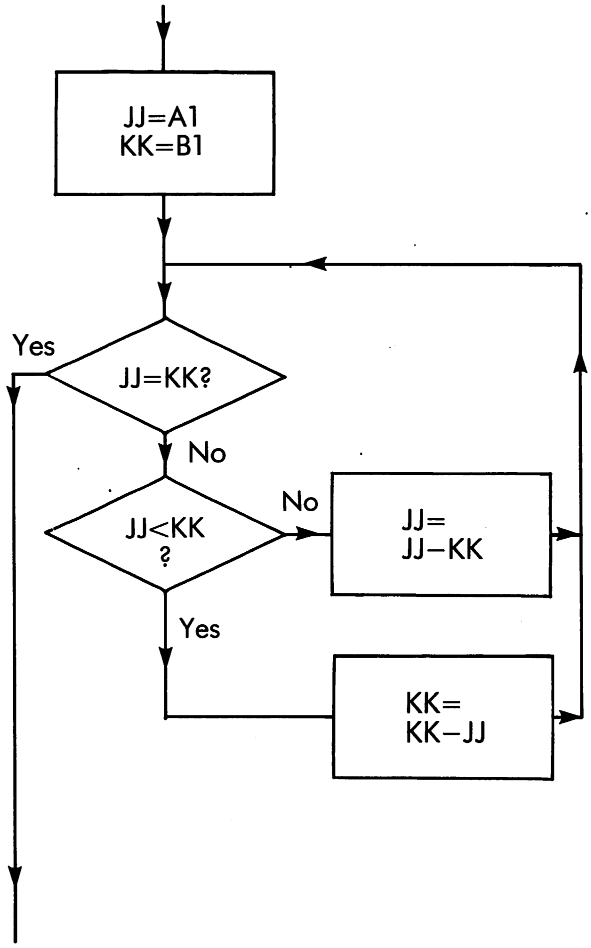
A1/B1: Fraction to be reduced to its lowest terms
C1/D1: Result

A simple way of finding the HCF of two numbers is called Euclid's Algorithm. Starting with the two numbers themselves subtract the smaller from the larger until the two are the same: this value is the HCF. Starting with 10 and 24 we get:

| | | |
|---|---|---|
| 24 | 10 | |
| | | Take 10 from 24 |
| 14 | 10 | |
| | | Take 10 from 14 |
| 4 | 10 | |
| | | Take 4 from 10 |
| 4 | 6 | |
| | | Take 4 from 6 |
| 4 | 2 | |
| | | Take 2 from 4 |
| 2 | 2 | |
| | | 2=2, so HCF of 10 and 24 is 2 |

This process can be flow-charted as follows:

*Note: We use local variables JJ and KK so as not to spoil the values of A1 and B1.*




**Glossary**

A1, B1: Number whose HCF is wanted
JJ, KK: Local variables (result in JJ)

The subroutine could look like:

```
5500 REM REDUCE FRACTION A1/B1 TO
        ITS LOWEST TERMS
5510 REM RESULT IN C1/D1 LOCALS ARE
        JJ,KK
5520 JJ=A1 : KK=B1
5530 IF JJ = KK THEN 5560
5540 IF JJ < KK THEN KK=KK−JJ : GOTO
        5530
5550 JJ=JJ−KK : GOTO 5530
5560 C1=A1/JJ : D1=B1/JJ
5570 RETURN
```

The last remaining details of the specification can now be filled in:

**Subroutine Specification**

*Purpose:* To simplify fractions to their lowest terms

*Line numbers:* 55ØØ to 557Ø

*Parameters:* Input A1 (top of fraction)
B1 (bottom of fraction)
Output C1 (top of simplified fraction)
D1 (bottom of simplified fraction)

*Local Variables: JJ, KK*

**DRIVER PROGRAM**

To check the subroutine, we need a 'driver' program. This is the simplest 'main program' we can construct which tests the subroutine in every reasonable way.

The specification of the subroutine turns out to be extremely useful in writing the driver program. It tells us

(a) To put the input parameters in A1 and B1
(b) To call the subroutine at 55ØØ
(c) To look for the results in C1 and D1
(e) To avoid using lines 55ØØ to 557Ø for any other purpose
(f) Not to use JJ and KK in the main program.

In general, if a specification doesn't include the information you need to write a driver program, the specification is incomplete.

A suitable driver program is this:

```
1Ø INPUT "FRACTION"; A1,B1
2Ø GOSUB 55ØØ
3Ø PRINT "RESULT ="; C1; "/"; D1
4Ø GOTO 1Ø
```

A few tests produce the following:

```
RUN
FRACTION? 33, 67
RESULT = 33/67
FRACTION? 33, 69
RESULT = 11/23
FRACTION? 12345, 23456
RESULT = 12345/23456
FRACTION? 1Ø, 24
RESULT = 5/12
FRACTION? 3, 6
RESULT = 1/2
....
```

These results look hopeful, and for the moment we accept the subroutine as being correct.

# EXPERIMENT 19·1

Write a program which lets the user type in *two* fractions (say p/q and s/t) and displays the simplified result of adding them together. For instance:

```
FIRST FRACTION? 3,8          (meaning 3/8)
SECOND FRACTION? 5,12        (meaning 5/12)
SUM = 19/24
```

HINT: $p/q+s/t = \dfrac{p\star t + q\star s}{q\star t}$.

Set A1=P★T+Q★S, B1=Q★T, and then call up the simplifying subroutine.

| *Experiment 19.1 Completed* | |
|---|---|

## SUBROUTINE ROBUSTNESS

It is interesting to compare programs written by professionals with those produced by inexperienced amateurs. A professional's program has two vital aspects:

(a) It is *robust*. It never gives wrong answers, and it never 'collapses' by displaying rubbish or getting stuck in a loop if the user gives it incorrect information.

(b) It is *adaptable*. The program is constructed and documented with flow charts, glossaries, subroutine specifications and REMarks so that any competent programmer can easily alter it to fit a new requirement.

In contrast an amateur's program is *fragile*. It usually works so long as the writer is standing by, ready to intervene if any incorrect input is keyed in. It isn't documented at all, and probably has no discernible structure. A few months after writing such a program, the programmer forgets how it works, and then no one understands it any more. Under these conditions most attempts to correct any remaining errors or improve the program simply make matters worse.

A chain is only as strong as its weakest link. In the same way a program is only as reliable as its least reliable subroutine. One of the most important ideas of software engineering is that every subroutine should be *perfect*, or at least as perfect as you can make it. It should be *impossible* for a subroutine to do anything wrong without at least giving some warning.

## LIMITING THE RANGE OF A PARAMETER

On page 174 there was a simple subroutine, with a single parameter X1, which displayed a number of stars on a line. In that unit we assumed, with optimism, that it was correct; but now let's examine it much more closely. Here it is again, together with a driver program:

```
  10 INPUT"NUMBER OF STARS"; X1
  20 GOSUB 3000
  30 GOTO 10
3000 REM DISPLAY NUMBER OF ★'S
       GIVEN IN X1 ON ONE LINE
3010 FOR JJ = 1 TO X1
3020 PRINT "★";
3030 NEXT JJ
3040 PRINT
3050 RETURN
```

First we notice that the local variable JJ isn't mentioned in the REMark at line 3000 and accept this is as a minor but genuine fault. Now we start testing. The subroutine seems to work well for X1=1, 3, 6, and so on. Feeling confident we try a few other numbers:

21: Works correctly.
22: Gives 22 stars and an extra blank line! If we were using the subroutine to draw a chart, this would spoil its appearance.
23: This *doesn't* give a line with 23 stars; it displays a line of 22 and a second line with one star.
0: We expect a blank line; but instead we get a line with *one* star.
–3: This is a nonsensical value, so we'd expect the subroutine to give a warning "What do you mean?". Instead, it gives a line with one star, just as if we had set X1=1.



It is becoming painfully clear that the program doesn't conform to its specification. We need some modifications:

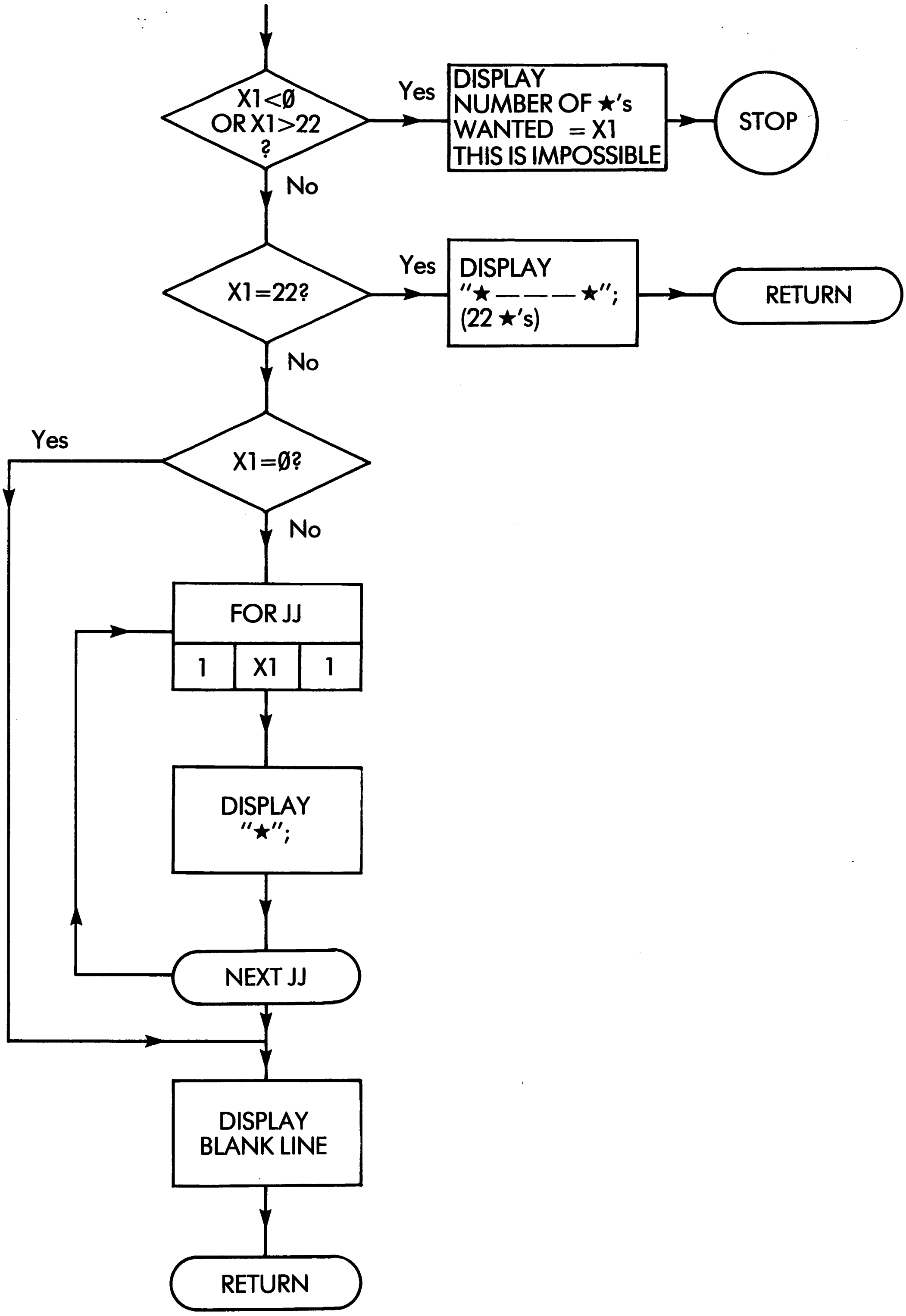
First, a line can only hold between 0 and 22 stars, so we should make the subroutine *reject* any value outside this range. A calling program which did supply an out-of-range value for the parameter X1 would presumably be in error, so it is appropriate to make the subroutine display a warning message and stop.

Second, the subroutine needs special provision for the extreme values 0 and 22. In the original version 0 was incorrectly handled because the FOR command in BASIC is always obeyed at least once, *even* for commands like

FOR JJ = 1 TO 0

At the other extreme, 22 led to an unwanted extra line because a new line is *automatically* inserted after the 22nd character in any line.

Taking these matters into account, we get a revised flow chart and program:




**Glossary**

X1: Number of stars to be displayed
JJ: Counter for stars

```
3000 REM DISPLAY STARS GIVEN IN X1 ON
     ONE LINE. MUST BE IN RANGE 0-22.
     LOCAL IS JJ.
3010 IF X1<0 OR X1>22 THEN PRINT "NO.
     OF ★'S WANTED="; X1 : PRINT
     "THIS IS IMPOSSIBLE" : STOP
3020 IF X1=22 THEN PRINT
     "★★★★★★★★★★★★★★★★★★★★
     ★★";:PRINT : RETURN
3030 IF X1=0 THEN 3070
3040 FOR JJ=1 TO X1
3050 PRINT "★";
3060 NEXT JJ
3070 PRINT : PRINT
3080 RETURN
```

This illustrates three important facts:

(a) Where the parameters of a subroutine can only take a certain limited range of values, good software engineering requires that the subroutine should check that every value is within range and report any discrepancies.

(b) When a subroutine is being tested, it is particularly important to try out the extreme allowable values (such as 0 and 22) since this is where errors often lurk.

(c) Subroutines which are properly engineered to be safe under all circumstances are usually longer than their simple-minded counterparts.

# EXPERIMENT 19·2



(a) The program below is meant to display the bowling records of 11 cricket players in the form of a "histogram" or chart, where each row stands for a player and each star for a wicket. Run the program both with the old and new versions of the subroutine starting at line 3000, and observe the difference:

```
 10 REM BATTING HISTOGRAM
 20 PRINT " SHIFT and CLR HOME "
 30 FOR J = 1 TO 11
 40 READ X1 : GOSUB 3000
 50 NEXT J
 60 STOP
100 DATA 3, 0, 15, 22, 5, 0, 4, 1, 0, 22,5
```

(b) Go back to the subroutine given on page 182, and test it again, more thoroughly. What happens if
A1 or B1 are 0 or negative (e.g. −5)?
A1 or B1 are decimals (such as 3.143)?

These tests will convince you (if you didn't already know) that Euclid's algorithm only works for positive whole numbers.

Design a properly engineered version of the subroutine, remembering that:

(a) It is not sensible for A1 and B1 to have fractional values (even though it could happen). If a number X is a *whole* number, the expression X=INT(X) is true.

(b) It is not sensible for B1 to have any value less than 1.

(c) It *is* sensible for A1 to be 0 or a negative number; this could arise during subtraction of two fractions. If A1=0, the value of the result should be 0/1, irrespective of B1. If A1 is negative your subroutine should remember the fact, use a *positive* number in Euclid's algorithm, and change the sign of C1 just before the result is delivered.

| *Experiment 19.2 Completed* | |
|---|---|

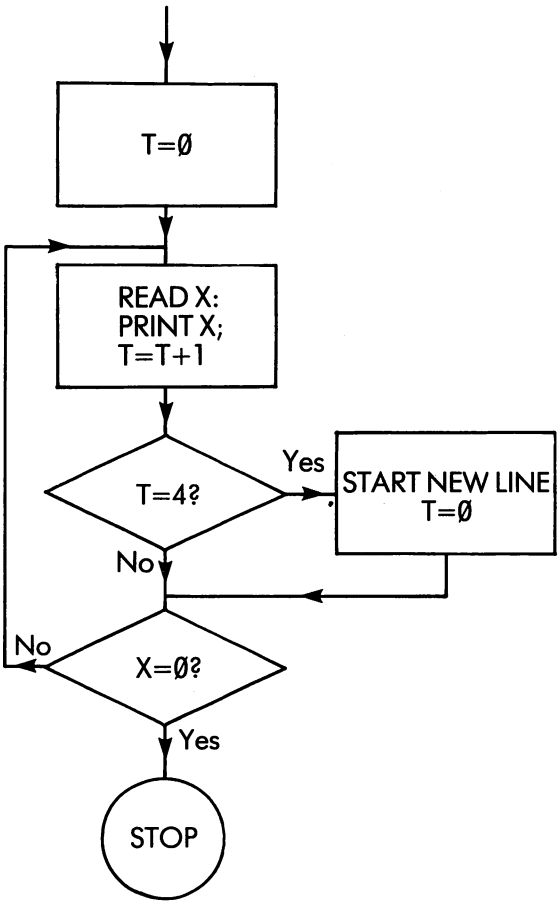
## NAMING CONVENTIONS IN SUBROUTINES

In discussing subroutines we've introduced a naming convention: A1, B1, . . . Z1 for parameters and AA . . . ZZ for local variables. There are no fixed rules about these names in BASIC, and you'll find plenty of programs which don't keep to the rule; but the convention is worthwhile because it protects you against some of the more subtle faults which can occur in large programs.

In practice, it is distressingly common for programs to fail because some vital variable has had its value spoiled by a subroutine. Here is a very simple example.

Consider a program which has a list of numbers in its data commands. It is supposed to read and display them four per line. The last number, which acts as a terminator, is zero.

To organise the layout, we use a variable T to count the number of numbers already on a line. Every time a new number is displayed we increase T by 1. When it reaches 4 we start a new line and return T to zero. Our general flow chart and program are:




**Glossary**

X: Current number
T: Number of items on current line

```
1Ø T=Ø
2Ø READ X
3Ø PRINT X;
4Ø T=T+1
5Ø IF T=4 THEN PRINT:T=Ø
6Ø IF X <> Ø THEN 2Ø
7Ø STOP
8Ø DATA 15,23,4Ø,11,37,51,99
9Ø DATA 33,12,89,53,17,2Ø,Ø
```

If you key in this program, you will find that it works perfectly.

Now suppose that a year later, when you have forgotten about the details of the program, you decide to make a modification: you want the computer to make a 'pip' sound every time it displays a new number. In your subroutine catalogue you find:

**Subroutine Specification**

*Purpose:* To make a 'Pip' followed by 1/2 second silence

*Lines:* 2ØØØ-2Ø5Ø

*Parameters:* None

This subroutine seems entirely suitable. You add its text to your program:

```
2ØØØ REM MAKE A PIP
2Ø1Ø POKE 36878,15: POKE 36876,245
2Ø2Ø FOR T=1 TO 1ØØ: NEXT T
2Ø3Ø POKE 36878,Ø: POKE 36876,Ø
2Ø4Ø FOR T=1 TO 5ØØ: NEXT T
2Ø5Ø RETURN
```

and insert a new command:

```
25    GOSUB 2ØØØ
```

Unfortunately, your program doesn't work any more; the pips come as you would expect, but the layout is all over the place. The reason is that the layout control variable T has been corrupted by the subroutine. This wouldn't have happened if the writer of the subroutine had followed even part of the conventions. If he'd called his local variable TT (instead of T) you wouldn't have used it in the main program; or if he'd mentioned 'T' as a local variable in the subroutine specification you'd have been warned.

In this example, the fact that the modified program was faulty was immediately obvious. Sometimes the error is not at all plain; it just leads to wrong answers. Consider this program which inputs a set of numbers and displays their total:

```
10 INPUT "HOW MANY NUMBERS"; N
20 T=0
30 FOR J=1 TO N
40 INPUT X
50 T=T+X
60 NEXT J
70 PRINT "TOTAL IS"; T
80 STOP
```

**Glossary**

N: Number of numbers
T: Running total
X: Next number to read
J: Count of numbers input

This works well. Now suppose that the programmer decides to improve matters by making the program give out a pip every time it accepts a number. He adds the subroutine from the catalogue, and splices in two extra commands:

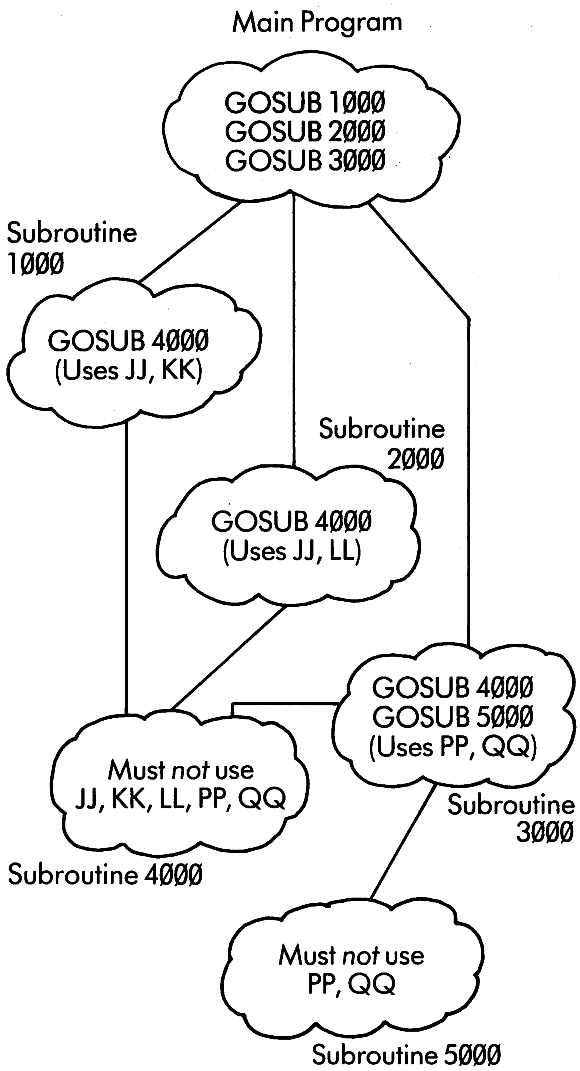
15 GOSUB 2000
and 45 GOSUB 2000.

The program is now much more satisfying to use: it sounds like a modern electronic cash register. Unfortunately, it now asserts that when you add up 247, 37, 12, 93, 52 and 39, you get 540. This answer looks reasonable, but is actually wrong! If you didn't notice the error and went on using the program in your business, you might end up as a case of "computer-assisted bankruptcy". The fault is easy enough to see once you know it's there, but the sad fact is that there are lots of similar faults tucked away in programs, completely unsuspected until they cause bridges to collapse, patients to die and rockets to crash into the sea.

If you follow the naming conventions you can usually avoid this type of fault. Your main program must *never* use "double letter" variables which are reserved for local variables in subroutines, and it should only use the "letter—1" form such as A1, or B1 for parameters.

If your program uses more than one subroutine, you have to be careful that they all fit together. Clearly all the subroutines must use different line numbers, and if necessary you will need to alter one or more of them accordingly.

When two or more subroutines are called "at the same level" (for example, they might both be called by the main program) they can safely use the same parameters and local variables. If the subroutines are at different levels and one of them 'calls' the other, they must use different local variables and parameters. The following diagram makes this clear:




The units which follow will make plenty of use of subroutines of all kinds. Get a loose-leaf ring binder and start your own subroutine library. Each entry should have four items of documentation:

Specification
Flow chart
Glossary
Source text (i.e. the program itself)

If you have the minimal VIC system, you will have to type the text of your subroutines into your programs by hand. More advanced systems allow you to keep the subroutine on a cassette tape or a floppy disk and copy them automatically.

189

# EXPERIMENT 19·3

Design, write and document a subroutine which takes three numbers as parameters, and delivers the value of the *largest* as its result. Write a suitable driver program and test your subroutine as thoroughly as you can.

| *Experiment 19.3 Completed* | |
|---|---|

# EXPERIMENT 19·4

The file called "BIGLETTERS" on the cassette tape is a subroutine which allows a user to type a letter or character and have it displayed four times normal size.

The full subroutine specification is shown below. Study the subroutine specification and write a driver program to create a banner headline.

**Subroutine Specification**

*Purpose:* To display VIC characters four times their normal size.

*Line Numbers:* 8000 to 8200.

*Parameters:* Input: The subroutine must be called once for each character. The character to be displayed should be supplied as a one character string in A1$.

Output: A1$ (converted to four times normal size).

*Local Variables:* AA, BB, JJ, KK, LL, MM, NN, QQ.

*Notes:* (i) QQ must not be used outside the subroutine for any purpose.

(ii) The subroutine handles all printable characters in the 'unshifted', 'shifted' and 'Commodore' sets. It also accepts and interprets BLK, WHT, RED, CYN, PUR, GRN, BLU, YEL, RVS ON, RVS OFF, CLR HOME and RETURN. Other keys such as DEL and Cursor control are ignored.

| *Experiment 19.4 Completed* | |
|---|---|

# UNIT: 20

## ARRAYS

When I was at school, our class list ran something like this:

ADAMS
BAXTER
COLIN
FINLAY
MCGREGOR
MCTAVISH
SMITH
THOMSON
WRIGHT
ZELL

I was fortunate in having a name which begins with a letter so near the beginning of the alphabet. It meant that I was among the first to be offered a choice of desk, etc., and I didn't have to wait long for my interview with the Careers Master. Poor Zell was never given any choice at all, and was always last in every queue. Sometimes we used to appeal to the teacher to turn the list round and let Zell be first for a change, but he never agreed. The idea was probably too complicated.

Let's think about writing a program to turn round a list of names. We want to let the user type in a class list, one name per line, and then read back the list from the screen in reverse order. On the face of it, this doesn't seem a very difficult task compared to the ones we have already programmed; and yet the solution is elusive. The best we can do is to find out in advance *how many* names there will be, and then write a long and clumsy program, using a different variable for each name. For instance, if there are four names in the list, we select variable A$,B$,C$ and D$, and write the following:

```
 1Ø PRINT "ENTER NAMES OF PUPILS"
 2Ø INPUT A$
 3Ø INPUT B$
 4Ø INPUT C$
 5Ø INPUT D$
 6Ø PRINT "REVERSED ORDER IS"
 7Ø PRINT D$
 8Ø PRINT C$
 9Ø PRINT B$
1ØØ PRINT A$
11Ø STOP
```

This program will turn round a list of exactly four names, but it cannot be adapted to work for any other number of names without adding (or deleting) extra commands. If the class has — say — 30 pupils, we would need to write a special program with 30 variables and 63 commands. Writing the program would be like a school punishment, and it would be much easier to turn the list round by hand.

Fortunately, BASIC has an important mechanism which helps us overcome this difficulty: it is called the *array* facility.

## DIMENSION STATEMENTS

An array is a *family* of variables, sharing the same "surname" but having individual "first names" called *subscripts*. If we want to use such a family, we normally tell the computer about it in a special command called a DIMension statement, thus:

```
This can be the name of any variable
         (The 'SURNAME')
   ↓
DIM W$ (5)
        ↑
  This can be any whole number
       (The 'FIRST NAME')
```

This tells the VIC to set aside space for a family of string variables called W$. There are *six* of them, and their full names are:

W$(Ø)
W$(1)
W$(2)
W$(3)
W$(4)
W$(5)

The subscript is the number in brackets which follows the family name of the array. The first variable has a subscript of Ø, so that the number of variables in the family is always *one more* than the number in the DIM statement. It is sometimes convenient to forget about the presence of the variable with subscript Ø, and to use only the variables which have subscripts starting at 1.

## USING ARRAY VARIABLES

In most ways the members of a family of variables are just like ordinary variables. You can include them in expressions, print, read and input them, and assign them values. If the family name ends with $, then each member can hold a string; otherwise, each member holds a number.

To illustrate these points, here are some legal BASIC commands. They are not intended to form a sensible program!

```
DIM N(2Ø):REM DECLARES AN ARRAY OF
               21 ELEMENTS CALLED N(Ø)
               TO N(2Ø)
N(5)=N(3) +5
PRINT N(1); N(2); N(3)
INPUT N(2)
IF N(12) = N(17) THEN 15Ø
```

Note that one thing you *can't* do is to use a member of a family, an 'array element' as it is often called, as the controlled variable in a FOR command.

```
FOR N(5) = 1 TO 17  }
. . .               }   Is not allowed
NEXT N(5)           }
```

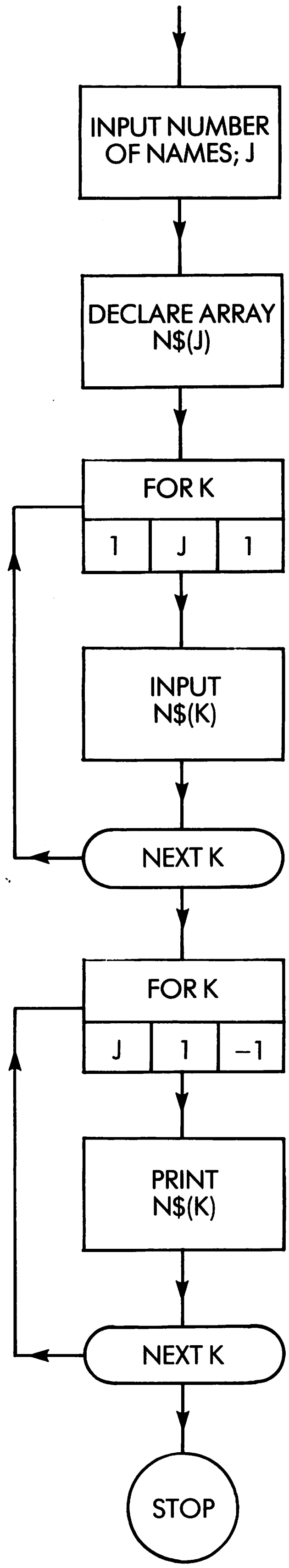
So far, we haven't said anything that would seem to help with the problem of inverting the name list. Here is the key point:

> The subscript of an array element may be an expression which is worked out as the program is running.

Think about this idea for a moment, and see if you can spot some of the implications before reading on.

Consider a command which is part of a loop, so that it is obeyed several times over. If the command includes a reference to an array element, then you can choose a subscript expression so that a different element is used every time round the loop!

You can now write a much more satisfactory solution to the original problem. The only extra requirement is that you ask the *user* to start by giving the number of names in the list.




**Glossary**

J: Number of names
N$(1) to N$(N): List of names
K: Name of counter and index to N$

The corresponding code can be written:

```
1Ø INPUT "HOW MANY NAMES";J
2Ø DIM N$(J)
3Ø FOR K=1 TO J
4Ø INPUT N$(K)
5Ø NEXT K
6Ø FOR K=J TO 1 STEP −1
7Ø PRINT N$(K)
8Ø NEXT K
9Ø STOP
```

This simple program has achieved the generality which we found absent from the earlier attempts. It will work for any number of names from one up. Key it in and try it out. Notice that the array N$ isn't declared until the program 'knows' how many elements it must have. Then (forgetting about N$(Ø)), it is given exactly the right number of elements.

One thing you must never do is to declare the same array more than once. You must avoid sequences like

```
3Ø DIM A(5Ø)
. . .
7Ø DIM A(5Ø)
```

but you must also be sure not to put the array declaration inside a loop. For instance, if you tried to make the simple list-inverting program into a loop by putting

```
9Ø GOTO 1Ø
```

it would give a fault the second time it tried to obey the DIM command in line 2Ø.

# EXPERIMENT 20·1

Imagine that the class size is so big that the teacher can't be expected to count the number correctly. Write a version of the list reversing program that looks for the special terminator "ZZZZ" at the end instead of asking for the number at the beginning. For example, if the input is

```
CRANACH
DURER
MICHAELANGELO
TURNER
ZZZZ
```

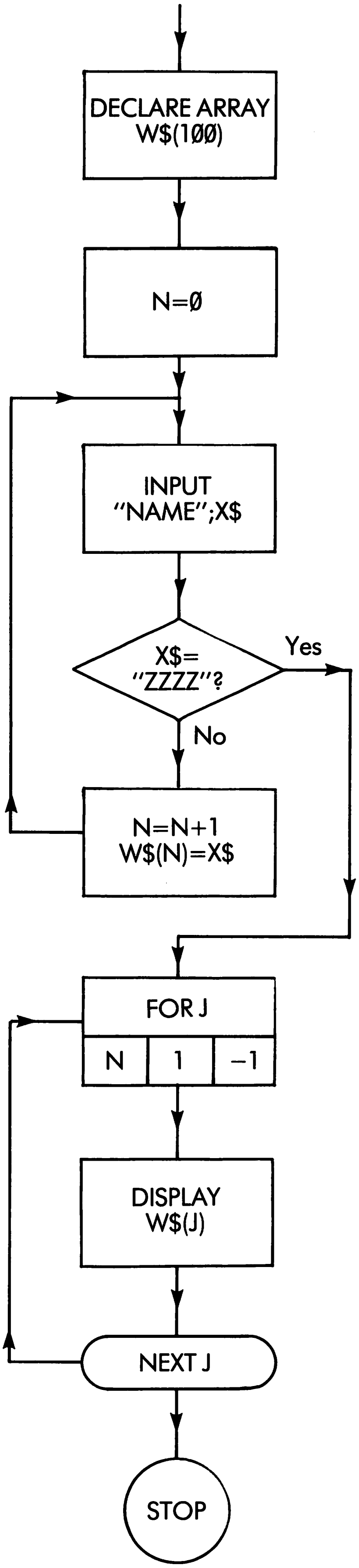
the output would be

```
TURNER
MICHAELANGELO
DURER
CRANACH
```

Hint: Use the following flow chart:

**Glossary**

W$(1ØØ): Array for names (max 1ØØ)
N: Count of names
X$: Current name
J: Index of W$




| | |
|---|---|
| *Experiment 20.1 Completed* | |

## FURTHER USES OF ARRAYS

As you will have seen from this example, one of the main advantages of the array facility is that it allows you to have a table of strings (or numbers) and to refer to its elements at any time and in any order. This is often useful. Let's imagine you are writing a program which has to display its results in words (such as "EIGHT" or "SEVENTEEN") rather than figures such as 8 or 17. We'll assume that all the results are known to be between 0 and 20. You can set up a table — we'll call it T$ — to translate figures into words. You arrange that each element contains the name of its own subscript, so that T$(0) = "ZERO", T$(1) = "ONE", and so on up to T$(20) = "TWENTY". Then to display any number X you simply put



```
PRINT T$(X)
```

For instance, if X=8, the command displays T$(8) which is the string "EIGHT".

Of course you have to do some work at the beginning of the program to get the table set up. You could always write a long list of 21 commands like:

```
T$(0)="ZERO"
T$(1)="ONE"
T$(2)="TWO"
. . .
T$(20)="TWENTY"
```

but it is less trouble to put the names of the numbers into a DATA statement and READ them in with a FOR loop. Your program would start:

```
10 DIM T$(20)
20 FOR J=0 TO 20
30 READ T$(J)
40 NEXT J
50 DATA ZERO,ONE,TWO,THREE,FOUR,
   FIVE
60 DATA SIX,SEVEN,EIGHT,NINE,TEN
70 DATA ELEVEN,TWELVE,THIRTEEN,
   FOURTEEN
80 DATA FIFTEEN,SIXTEEN,SEVENTEEN
90 DATA EIGHTEEN,NINETEEN,TWENTY
```

Let's use array T$ to display a multiplication table in words. A simple program, which forms the starting point for our design, is:

```
100 FOR J=0 TO 10
110 PRINT "2★";J;"=";2★J
120 NEXT J
```

We now modify the PRINT command by making it display the appropriate table entry instead of each number.

```
"2"   becomes T$(2)
"J"   becomes T$(J)
"2★J" becomes T$(2★J)
```

We get

```
100 FOR J = Ø TO 1Ø
11Ø PRINT T$(2);" ★ ";T$(J);" = ";T$(2★J)
12Ø NEXT J
```

If you key in these instructions following the ones labelled 1Ø-9Ø, you can try the program out for yourself.

A basic property of an array is that if you know the subscript of an element, you can select the element and bring it out straight away. Sometimes you want to go the other way; you know the value of the element, and you want to find out where (if anywhere) it occurs in the array. This operation is harder, since you have to make the computer search down the array, entry by entry, until it either finds one which matches your element, or else reaches the end of the array.

Let's take a simple example. You aim to write a program which inputs two numbers, adds them up and displays their sum, but communicates with the user entirely in *words*. For instance, a typical dialogue might be

```
GIVE TWO NUMBERS
?EIGHT,FIVE
SUM IS THIRTEEN
```

Both the words must be converted to numbers before they can be added together. The conversion from words to numbers occurs twice, and is a clear choice for a subroutine. The specification and code for the subroutine can be written down quite easily: they are

**Subroutine Specification**

*Purpose:* To convert a word into a number in the range Ø-2Ø

*Line numbers:* 1ØØØ-1Ø6Ø

*Parameters:* Input: Word to be converted
Output B1: Value of number

*Locals:* JJ

*External Reference:* T$(Ø-2Ø): Names of the numbers

```
1ØØØ REM CONVERT WORD A1$ INTO
     NUMBER B1
1Ø1Ø FOR JJ=Ø TO 2Ø
1Ø2Ø IF A1$ = T$(JJ) THEN 1Ø5Ø
1Ø3Ø NEXT JJ
1Ø4Ø PRINT "NO ENTRY FOUND": STOP
1Ø5Ø B1=JJ
1Ø6Ø RETURN
```

Note that the subroutine sets up a FOR loop to search down the list T$. It matches the given word in A1$ with T$(Ø),T$(1), and so on. When it finds a corresponding entry it jumps out of the loop to command 1Ø5Ø. If it searches all the way down the list and doesn't find an exact match, it prints a warning and stops.

The main program is straightforward. Here it is, with some extra comments:

```
 1Ø DIM T$(2Ø)
 2Ø FOR J=Ø TO 2Ø
 3Ø READ T$(J)
 4Ø NEXT J
 5Ø DATA ZERO,ONE,TWO,THREE,FOUR,
    FIVE
 6Ø DATA SIX,SEVEN,EIGHT,NINE,TEN
 7Ø DATA ELEVEN,TWELVE,THIRTEEN,
    FOURTEEN
 8Ø DATA FIFTEEN,SIXTEEN,SEVENTEEN
 9Ø DATA EIGHTEEN,NINETEEN,TWENTY
1ØØ PRINT "GIVE TWO NUMBERS"
11Ø INPUT X$,Y$
12Ø REM SET UP PARAMETERS AND CALL
    SUBROUTINE TO CONVERT X$ TO X
125 A1$ =X$: GOSUB 1ØØØ:X=B1
13Ø REM SAME FOR Y
135 A1$=Y$: GOSUB 1ØØØ:Y=B1
14Ø Z=X+Y: REM ADD THE TWO NUMBERS
15Ø IF Z > 2Ø THEN PRINT "RESULT NOT IN
    LIST": STOP
16Ø PRINT "SUM = "; T$(Z)
17Ø STOP
```

It's worth noting that we keep all the details of each subroutine call to one line. This includes setting up the input parameter, the actual call command, and extracting the result from the output parameter.

Command 15Ø is included because the program can't display any number higher than 2Ø. If the command were not there, and the user typed — say — TWELVE and FIFTEEN, then the machine would try to access T$(27). This element doesn't exist, and the VIC would come up with:

```
? BAD SUBSCRIPT
  ERROR IN 16Ø
```

In our version the machine still doesn't give the right answer, but at least the comment is a little more informative.

# EXPERIMENT 20·2

a) Modify the program in the last section so that it deals with ROMAN numbers up to XL (4Ø).

b) The data statements of a program contains 2Ø names and telephone numbers, arranged like this:

```
DATA MAXWELL, 3398123
DATA BOHR, 558
DATA EINSTEIN, 4073189
DATA VON NEUMANN, 777000
DATA NEWTON, 3074
DATA ZUSE, 222
DATA PLANCK, 1237543
DATA BOYLE, 146543
DATA BABBAGE, 03474
DATA LAPLACE, 5674
DATA PTOLEMY, 54863
DATA ARISTOTLE, 66543
DATA MCCARTHY, 47
DATA DIJKSTRA, 645
DATA BERZELIUS, 777
DATA CHARLES, 5543
DATA MENDELEEV, 645634
DATA TSIOLKOVSKY, 645332
DATA ARCHIMEDES, 2
DATA HOYLE, 21352
```

Design and write a program which invites the user to type a name, then looks the name up in the directory, and displays the corresponding telephone number if found. If not found, the program should display a suitable message. Two typical runs might be:

```
NAME? NEWTON
NEWTON'S PHONE IS 3074
                              Typed by User
NAME? FREUD
FREUD HAS NO LISTED
PHONE NUMBER
```

*Experiment 20.2 Completed* ☐



The self test quiz for this unit is entitled UNIT2ØQUIZ.

# UNIT: 21

## STRING FUNCTIONS

String handling is a vital feature of the BASIC language, and gives it the power to solve problems in almost every area of daily life. So far, however, the programs we have considered have all taken strings as complete indivisible objects. Every string was stored, moved around and displayed in exactly the same form as it was originally keyed into the VIC.

In this unit we shall be looking at some special functions which allow you to break up strings into small sections, and even into individual characters. These functions help you solve all kinds of problem which would otherwise be difficult or impossible. For instance, you'll be able to extract the surname from a person's full name, and you'll learn how to get the VIC to display long sentences so that no words are spread over more than one line.

The functions involved are called 'string functions' because they either use or generate strings rather than numbers. Any function which generates a string as its result has a $ sign as the last character of its name such as MID$, STR$, and so on.

### The "LEN" String Function

The string functions are built into BASIC, so you can try them out directly, without even including them in a program. Let's try some. Switch on your machine and type the following lines ending each line with the RETURN key.

```
PRINT LEN("VIC")
PRINT LEN("COMMODORE")
Z$= "STRING"
PRINT LEN(Z$)
```

In each case, the VIC displays the LENgth of the string involved. In general, the LEN function delivers the number of characters in the argument string. In this context 'argument' is a technical word which means the object on which a function is used.

Notice the way LEN is written:

LEN (argument string)

The argument must be enclosed in brackets. It can be an explicit string enclosed in quotes, or the name of a string variable, or any expression which produces a string as its result.

The LEN function produces a *number*, so the whole construction can be used wherever a number is needed. For instance you might see

```
X=LEN(Q$)
```

or `FORJ=1 TO LEN (P$)`

or `PRINT LEN("BUY" + S$ + "LOAVES")`

### The "MID$" String Function

Another vital function is MID$. This function selects a *portion* of any string it is given for its argument. Type the command

```
PRINT MID$("ABCDEFG",2,4)
```

The result shows you how MID$ works. In this case it displays a *4* character string, starting at the *2nd* character of "ABCDEFG".

In formal terms, the MID$ function takes three arguments which are separated by commas and enclosed in brackets. The arguments are as follows:

The first is the string to be used.

The second is a number specifying the position of the first character in the result.

The third is another number giving the length of the result.

As you would expect, any of the arguments can be variables of the appropriate sort. The length of the result can be anything from Ø (called the 'null' string) to the full length of the first argument. In practice it is often one character.

Here is a simple program to input a word and display it backwards. Study it carefully and note how the functions LEN and MID$ are used:

```
1Ø INPUT "PLEASE TYPE A WORD"; X$
2Ø PRINT "YOUR WORD BACKWARD IS"
3Ø FOR J = LEN(X$) TO 1 STEP -1
4Ø PRINT MID$(X$,J,1);
5Ø NEXT J
6Ø STOP
```

Key the program in and check it for yourself; try out words of 1, 2 or more characters.

## EXTRACTING SURNAMES

Now let's move on to extracting larger portions of strings. If you ask someone to type their full name, they might use any of the following forms:

J.P.JONES

or JANET BLOGGS

or GEORGE P O'HAGAN

or ALFRED HENRY FFOULKES-SMYTHE

If you want to extract the surname from such a string, it is no good working from the front, because the computer can't tell a surname from a second Christian name or even a string of initials. However, the surname always comes last, and this suggests a way of locating it. Examine each character starting from the end of the string until you come to one which can't be part of a surname. The next position to the right must be where the surname starts. If *every* character in the string is part of the surname, its owner is clearly from a country like Afghanistan, where people only have one name.

Look at this string which has the positions of the characters marked:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| J | . | P | . | J | O | N | E | S |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

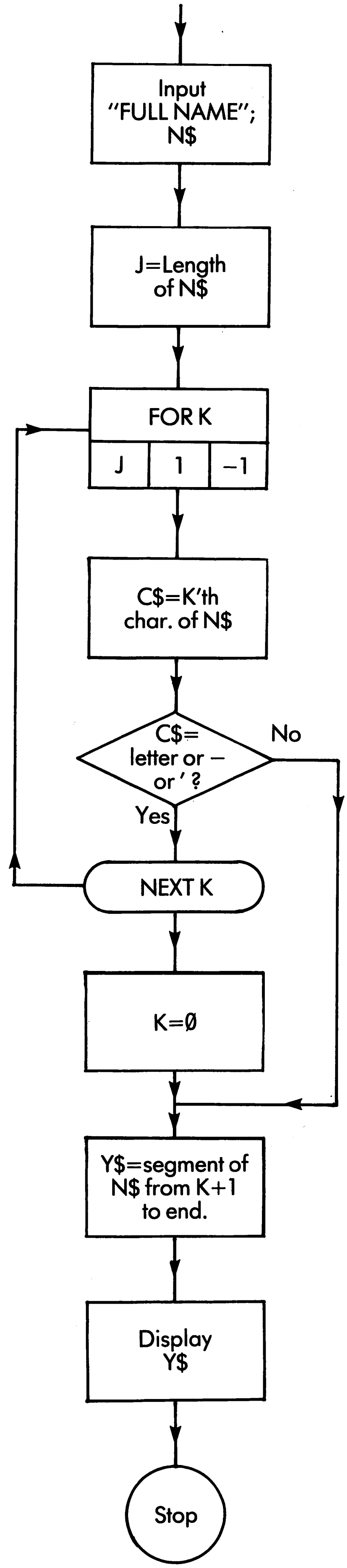
If you search from the right, the first character you come to which can't be part of a surname is the full stop in position 4. The surname therefore starts at position 5, and can be extracted by a MID$ function:

MID$ (N$,5,5) (where N$= "J.P.JONES")
gives "JONES".

In general, if the length of the whole string is J, and the position of the first character not in the surname is K, the surname itself will start at character (K+1) and its length will be (K−J).

What symbols can a surname include? The examples suggest that letters, the hyphen and the apostrophe are the only characters we need to expect.

Now we have collected enough ideas to sketch out a flow chart. It goes like this:




**Glossary**

N$: String with full name
J: Length of N$
K: Used to scan backwards along the string
Y$: Result: surname in N$

Next we can try out the algorithm in a short program, thus:

```
 1Ø INPUT "FULL NAME"; N$
 2Ø J=LEN(N$)
 3Ø FOR K=J TO 1 STEP -1
 4Ø C$=MID$(N$,K,1)
 5Ø IF NOT(C$>="A"AND C$<="Z"
    OR C$="-"OR C$="'")THEN 8Ø
 6Ø NEXT K
 7Ø K=Ø
 8Ø Y$=MID$(N$,K+1,J-K)
 9Ø PRINT Y$
1ØØ GOTO 1Ø
```

*Comments:* C$ is used to hold the K'th character of the full name. It is part of a surname if:

(a) it is a letter (i.e. it lies in the range A-Z),

or (b) it is a hyphen,

or (c) it is an apostrophe.

The conditional statement jumps to 8Ø if C$ is *not* part of a surname.

Line 7Ø is only executed for people with one name. When the FOR command ends, the controlled variable (K in this example) is "undefined" (which means it may have any value whatever) and not necessarily Ø. Therefore K must be set so that line 8Ø can be obeyed.

The command in line 8Ø extracts the surname and puts it in Y$.

When this program is keyed in, it seems to work correctly on all the examples supplied. The program fulfils a generally useful function, so we make it into a *subroutine* with the following specification and code. Note the change in variable names:

**Subroutine Specification**

*Purpose:* To extract a surname from a full name

*Lines:* 41ØØ-418Ø

*Input parameter:* N1$ contains a full name

*Output parameter:* Y1$ delivers the surname

*Local variables:* JJ, KK, CC$

```
41ØØ REM EXTRACT SURNAME FROM N1$
     AND DELIVER IT IN Y1$
411Ø JJ=LEN (N1$)
412Ø FOR KK=JJ TO 1 STEP -1
413Ø CC$=MID$(N1$,KK,1)
414Ø IF NOT (CC$>="A" AND CC$<="Z"
     OR CC$="-" OR CC$="'") THEN
     417Ø
415Ø NEXT KK
416Ø KK=Ø
417Ø Y1$=MID$(N1$,KK+1,JJ-KK)
418Ø RETURN
```

A driver program to test out this subroutine would be:

```
1Ø INPUT "NAME PLEASE"; N1$
2Ø GOSUB 41ØØ
3Ø PRINT "SURNAME IS  "; Y1$
4Ø GOTO 1Ø
```

## USING MID$ TO AMEND A STRING

A final point about the MID$ function: you cannot use it on the left side of an assignment command. For instance, if you want to change the fourth character of string X$ into a "U", you may *not* write

MID$(X$,4,1) = "U" ← not BASIC

However, you can accomplish the same thing by splitting the string up into three portions and recombining them with the + operation:

$$X\$ = \underbrace{\text{MID\$(X\$,1,3)}}_{\text{First 3 characters of X\$}} + \text{"U"} + \underbrace{\text{MID\$(X\$,5,LEN(X\$)}-4}_{\text{End of X\$ for character 5 onwards}}$$

# EXPERIMENT 21·1

This experiment is in three parts:

(a) Write a program which inputs a string from the keyboard and displays it, first having changed every "E" to an "O". The output might be:

> This idoa was takon from a TV program foaturing Ronnio Barkor among othor pooplo.

(b) You can apply the MID$ function to the 'time' variable TI$ so as to extract the hours, minutes and seconds as separate 2-character strings. Write a short program which displays the current time thus:

13/23/57

(c) The surname extraction subroutine suffers from a fault which we didn't find in our original tests: if someone types a full stop or a space after their surname the subroutine returns a null string. Design a suitable modification for the subroutine.

| *Experiment 21.1 Completed* | |
|---|---|

## LEFT$ AND RIGHT$

Two string functions which are often useful are LEFT$ and RIGHT$. As you can deduce from its name LEFT$ extracts the left-hand side of a string, and RIGHT$ the right side. Each function takes two arguments; the first (as in MID$) is the string to be partitioned, and the second is the length of the result. Thus

PRINT LEFT$("ABCDEFG",3) gives ABC

and PRINT RIGHT$("ABCDEFG",2) results in FG.

You will have noticed that neither of these two functions achieves anything which could not be done with MID$, but they are sometimes more convenient to use.



## POSITIONING THE CURSOR

One particular application of LEFT$ is to position the cursor at any point in the screen. We begin by setting up two string variables:

Y$ as a "HOME" followed by lots of "cursor down characters"

X$ as lots of "cursor left" characters:

```
10 Y$ = " [CLR HOME] [CRSR ↓] < 22 times > [CRSR ↓] "
20 X$ = " [CRSR ⇒] < 21 times > [CRSR ⇒] "
```

To move the cursor to a position Y lines down from the top of the screen, we arrange to PRINT the first (Y+1) characters of Y$: a "home" and cursor down Y times. Similarly we move X places across by PRINTing the first X characters of X$. These can be combined in a single statement:

```
100 PRINT LEFT$(Y$,Y+1); LEFT$(X$,X);
```

## PERMUTATIONS — n!

The next example deals with permutations. Permutations are useful in solving problems and in finding anagrams for crosswords, and in a more serious vein they play an important role in statistics and in the design of scientific experiments. The section contains some easy mathematics, but if you find mathematics difficult you can just use the permutation program without reading the explanation. If the whole concept overwhelms you, skip over this part. Permutations are not an essential part of BASIC programming.

A *permutation* is a particular order of arranging a set of objects or events. For example, the order in which a peal of eight bells is rung is a permutation, and so is the order in which the horses in a race pass the winning-post (assuming there are no dead-heats).

Howmany permutations can you get? That depends on the number of objects. In a race with only one horse (a "walk-over") there can be only *one* outcome. If there are two horses called A and B, then either one of them can win, but the other must come second: there are *two* permutations AB and BA. With three horses, there can be three different winners, and in each case one of the two remaining horses can be the runner-up. There are *six* permutations: ABC, ACB, BAC, BCA, CAB and CBA. With four horses there can be 4 x 3 x 2 or *twenty-four* different results. The table shows the way these figures are going:

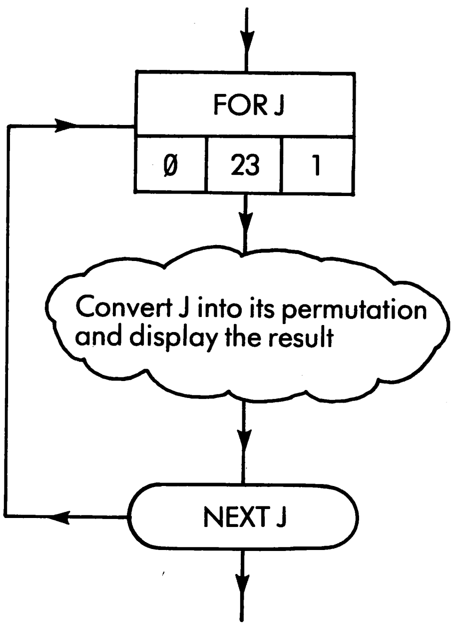
| *Number of Objects* | *Number of Permutations* | |
|---|---|---|
| 1 | | 1 |
| 2 | 2 = | 2 |
| 3 | 3x2 = | 6 |
| 4 | 4x3x2 = | 24 |
| 5 | 5x4x3x2 = | 120 |
| 6 | 6x5x4x3x2 = | 720 |
| 7 | 7x6x5x4x3x2 = | 5040 |
| 8 | 8x7x6x5x4x3x2 = | 40320 |

As you can see the number of permutations grows very quickly, and is well over three million for 10 objects. The number of permutations of 'n' objects is called "factorial n", a phrase which mathematicians sometimes write as "n!" to indicate the product of all the numbers from 1 to n multiplied together.

We shall develop a program which reads any string and displays all its permutations. For instance, if the input is 'TEA', the output should include TEA, TAE, ATE, AET, EAT and ETA.

Suppose the starting string is n letters long. We know that there will be n! (factorial n) different permutations, but how can we get the computer to work them all out without repeating itself or missing any out?

One way to tackle this problem is to invent a method of converting the numbers Ø, 1, 2, 3... into permutations so that each one is different from all the others. Then if n is — say 4 — (a four-letter string) we can produce all the 24 permutations of four letters by converting each of the numbers Ø to 23 into its corresponding permutation and displaying the results. The flow chart for such a program would be:




In this flow chart, we can call J a 'permutation number'. We still need to find a way to convert the permutation number into its corresponding permutation of letters. This problem isn't easy, but we might get some clues by looking at the answers in a few simple cases. Suppose the objects to be permuted are the letters A B C and so on.

1 Object: 1 Permutation

A

2 Objects 2 Permutations

AB
BA

3 Objects 6 Permutations

ABC ACB
BAC BCA
CAB CBA

4 Objects 24 Permutations

ABCD ABDC ACBD ACDB ADBC ADCB
BACD BADC BCAD BCDA BDAC BDCA
CABD CADB CBAD CBDA CDAB CDBA
DABC DACB DBAC DBCA DCAB DCBA

You will see that if each permutation is taken as a "word" then all the words of the same size are written out in dictionary order.

As you study these lists, a definite pattern emerges. Suppose we divide up the members in each set according to their first letters, then the permutations of *three* letters come in *three* groups of two each:

*ABC* *BAC* CAB
*ACB* *BCA* CBA

In each group the initial letter is followed by *all* the permutations of the other two. So A is followed by BC and CB. We call BC and CB 'sub-permutations'.

You can see this pattern carry over into the permutations of *four* letters. There are *four* groups, each with six members. Each initial letter is followed by the *six* sub-permutations of the other three.

For any permutation, we can define a 'group number' and a 'sub-permutation number'. The group number will indicate the first letter (according to the code A=Ø, B=1, C=2, and so on) and the sub-permutation number will be the position within the group (also starting at Ø). For instance, consider the permutation BCDA. This has a group number of 1 (because it starts with a B) and the sub-permutation number is 3, as you can check from the table above.

We now have a strong hint about a method of converting any permutation number into its corresponding permutation. We first find the group number, which settles the first letter; then we find the sub-permutation number and work out the corresponding permutation of the remaining letters!

To find the group and sub-permutation numbers, all we need do is to divide the permutation number by the size of the group. The quotient gives the group number and therefore the first letter, and the remainder specifies the sub-permutation number.

To give an example, consider permutation number 19 of four letters.

6)19

3 remainder 1


The corresponding permutation starts with D (D=3) and is followed by sub-permutation number 1 of the letters A B C.

The sub-permutation can be worked out by exactly the same process as the permutation itself. There are only two important changes:

1) The letter used at the front of the main permutation must be removed from the list of letters so that it is not used again.

2) The group size must be adjusted (say from 6 to 2, or from 2 to 1).

Let's give a specific example, taking permutation 9 of four letters. We begin by labelling the letters A=Ø,B=1,C=2,D=3.

1) We divide 9 by 6 and get quotient = 1, remainder = 3 . The first letter of the permutation is therefore *B*. We remove the B from the list of letters and relabel the others: A=Ø,C=1,D=2.

2) Now we find sub-permutation 3 from the letters A,C and D. The group-size is 2. Dividing 3 by 2 we get quotient = 1, remainder = 1 . The *next* letter of the permutation is therefore C. We remove the C from the list and relabel the other letters A=Ø,D=1.

3) Now we find sub-permutation 1 from the letters A and D. The group size is 1. Dividing 1 by 1 gives quotient = 1, remainder = Ø . The *next* letter of the permutation is D. We remove it from the list. This leaves only one letter, an A labelled Ø.

4) Finally we find sub-permutation Ø from the letter A. Obviously it is the A, but we can still use the same process as before: the group size is 1, and Ø divided by 1 gives quotient = Ø, remainder = Ø . As we expected the final letter is A, and the permutation as a whole is *BCDA*.

To satisfy yourself that you understand this process, try converting a few more numbers between Ø and 23, and make sure that your answers come out the same as the table above. (Remember that the first permutation ABCD corresponds to Ø, not 1.)

Now we'll convert the method into a program. The letters to be permuted are not necessarily ABCD, but can be anything the user types. Similarly the length of the string is arbitrary, although we can expect the program to run for a very long time if there are more than six or seven letters.

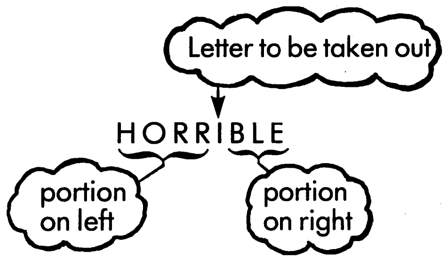
First, we have to maintain a 'pool' of letters and ensure that they are selected correctly. The simplest way of doing this is to put them in a string (say Y$). Since they are then numbered automatically the letter with 'notional' label Q can be selected as

MID$(Y$,Q+1,1)

The "+1" must be included because the automatic numbering scheme starts at 1, whereas our method produces group numbers starting at Ø.

## REMOVING LETTERS FROM A STRING

Once a letter has been used, it must be removed from the string. The others will then be moved up automatically, and this is equivalent to relabelling them. Taking a letter out of a string is quite easy: we concatenate (join together) the portion of the string to the left of the used letter and the portion to the right.

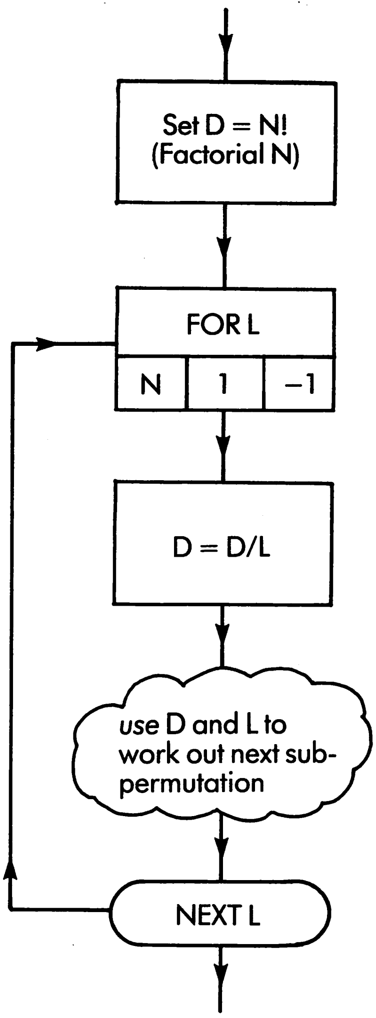



Result: HORRBLE

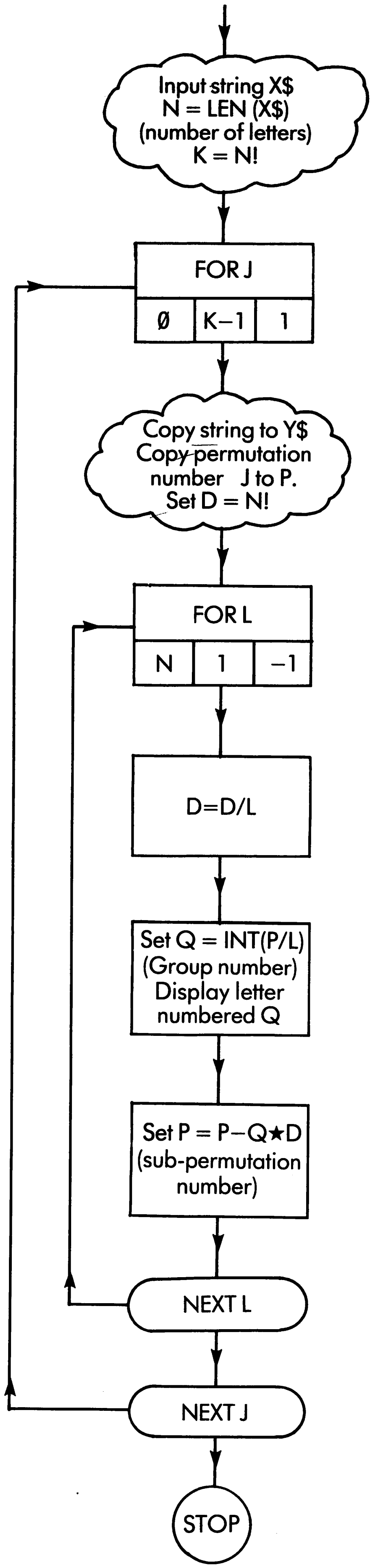
If the label number of the letter to come out is Q, then the left-hand portion will have Q letters, and the right-hand side (LEN(Y$) −Q−1). (Remember the labels run from Q to Q.) The required command is:

Y$ = LEFT$(Y$,Q) + RIGHT$(Y$,LEN(Y$) −Q−1)

Second, the number of letters in each sub-permutation go 4 3 2 1 and the corresponding group sizes go . . . 6 2 1 1. These numbers are the values of n! for different values of n, and can be produced by a program like this:






In this flow chart L and D will take the right sequences of values. For instance if N=6, then L will become 6,5,4,3,2,1 and D (by the time it is used) will be 12Ø,24,6,2,1 and 1.

We can now put all these ideas together into a program. The process of converting a string into a permutation gradually destroys it, so we need to keep a master copy of the original and replace the 'working copy' for each separate permutation. The flow chart and program are:

The Glossary is

X$: String to be permuted
N: Length of string to be permuted
K: Factorial N (calculated in line 3Ø)
J: Permutation number
Y$: Working copy of X$
D: Current group size
L: Number of letters in current sub-permutation
P: Current sub-permutation number
Q: Group number of current sub-permutation
S: Variable for all the numbers from 1 to N

After this complex analysis, the program turns out to be surprisingly short. It is:

```
 10 INPUT X$
 20 N= LEN(X$)
 30 K=1:FOR S=1 TO N:K=K★S:NEXT S
 40 FOR J = 0 TO K−1
 50 Y$ = X$: D = K: P=J
 60 FOR L = N TO 1 STEP −1
 70 D=D/L
 80 Q=INT(P/D): P=P−D★Q
 90 PRINT MID$(Y$,Q+1,1);
100 Y$=LEFT$(Y$,Q)+RIGHT$(Y$,LEN(Y$)
    −Q−1)
110 NEXT L
120 PRINT
130 NEXT J
140 STOP
```

The quotient and remainder are calculated in line 80. Remember that INT throws away any fraction; this is what makes the commands work correctly. For instance if P=17 and D = 6, then

Q = INT(17/6) = INT( 2.8333333) = 2
P = 17 − 6★2 = 17−15 = 5.

Key in this program and try it on your own data. See if you can modify it so that is displays more than one permutation on the same line.

## CONVERTING STRINGS TO NUMBERS — VAL

Two other string functions help to convert strings to numbers and vice versa.

VAL("a string")

takes a string of decimal digits (possibly preceded by + or −, and containing a decimal point) and converts it to the corresponding numerical value.

VAL is useful in getting valid input from very naive users. Consider a program which asks for a number to be typed, by an instruction such as

INPUT X

If the user actually types something which isn't a number, such as "PARDON", the BASIC system just says,

REDO FROM START

This isn't very helpful, and the puzzled user may not realise what is expected of him.

As an alternative, everything the user types can be read as a string, then there is no risk of a REDO FROM START message and the program can analyse the user's reply character by character, issuing suitable error messages if any mistakes are detected. Finally, if the string is found to consist of symbols which make up an acceptable number, the value can be extracted with VAL. Here is a specification, flow chart and subroutine for 'tolerant' input of numbers.

**Subroutine Specification**

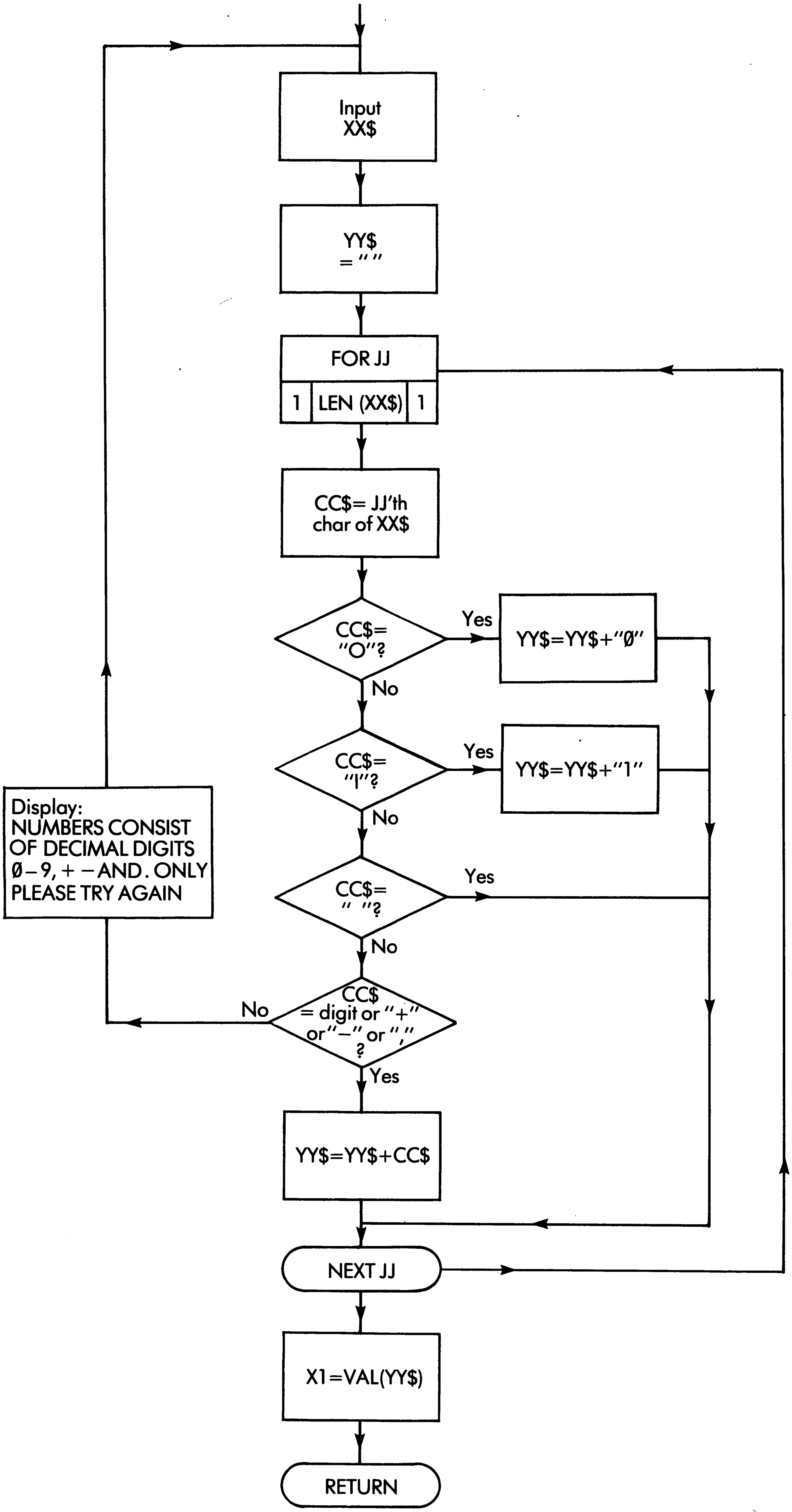
*Purpose:* To input numbers from an unskilled user.
All spaces are ignored, and letters I and O are taken as 1 and 0. Other errors are clearly explained.

*Lines:* 4500 to 4660

*Output parameter:* Result is delivered in X1

*Local variables:* XX$, YY$, JJ$, CC$

207



Input XX$
YY$ = ""
FOR JJ
1
LEN (XX$)
1
CC$= JJ'th char of XX$
CC$= "O"?
Yes
YY$=YY$+"Ø"
No
CC$= "l"?
Yes
YY$=YY$+"1"
No
CC$= " "?
Yes
No
CC$ = digit or "+" or "−" or "." ?
No
Display: NUMBERS CONSIST OF DECIMAL DIGITS Ø−9, + − AND . ONLY PLEASE TRY AGAIN
Yes
YY$=YY$+CC$
NEXT JJ
X1=VAL(YY$)
RETURN

```
4500 REM TOLERANT INPUT OF NUMBERS
4510 INPUT XX$
4520 YY$=""
4530 FOR JJ=1 TO LEN(XX$)
4540 CC$=MID$(XX$,JJ,1)
4550 IF CC$="O" THEN YY$=YY$+"0":
     GOTO 4600
4560 IF CC$="I" THEN YY$=YY$+"1":
     GOTO 4600
4570 IF CC$=" " THEN 4600
4580 IF NOT(CC$<="9" AND CC$>="0"
     OR CC$="+" OR CC$="-" OR
     CC$=".")THEN 4620
4590 YY$=YY$+CC$
4600 NEXT JJ
4610 X1=VAL(YY$):RETURN
4620 PRINT "NUMBERS CONSIST OF"
4630 PRINT "DECIMAL DIGITS 0-9,"
4640 PRINT " +, - AND . ONLY"
4650 PRINT "PLEASE TRY AGAIN"
4660 GOTO 4510
```

## CONVERTING NUMBERS TO STRINGS — STR$

STR$ (a number) works the opposite from VAL. It takes a number (or a numeric expression) as its argument and delivers a string of symbols, the same as those which would have been displayed if PRINT had been used.

STR$ is a valuable function for getting a neat layout of numbers on the screen.

The main trouble with the PRINT command is that you can never be sure what the exact layout is going to be. To illustrate this point, key in the following program:

```
 5 PRINT "NUMBER", "SQUARE"
10 FOR J=1 TO 7 STEP 0.1
20 PRINT J, J*J
30 NEXT J
40 STOP
```

Run this program slowly, holding down the CTRL key. At first, all seems well. The screen displays the table of squares you would expect. You get

```
NUMBER          SQUARE
 1               1
 1.1             1.21
 1.2             1.44
```

and so on.

However, the entry for 2.8 looks peculiar; it says

```
2.8             7.8399999
```

and is followed by a blank line. You know perfectly well that the square of 2.8 is 7.84, not 7.83999999 as it appears on the screen.

After 3.6, the table goes crazy. It reads:

```
3.6                     12.96
3.69999999
13.69
3.79999999
14.44
3.89999999
15.21
3.99999999
```

and so on.

The difficulty is due to two effects which interact with one another.

The first problem is that of 'truncation error'. Since the VIC — like most computers — works on the binary system, it can't handle decimals like 0.1 exactly. There is always a tiny error. In our program the value of J starts at 1 and grows towards 7 by repeated additions of 0.1; eventually the errors accumulate and show up in answers which are very nearly, but not quite what they are expected to be. Thus the difference between

7.84 and 7.83999999

is only 0.00000001, but this is enough to play havoc with the layout; the number *looks* quite different, and the extra blank line is forced because the last digit of the number falls into the last screen column.

The second problem emerges when the truncation error affects the printed value of J itself. "3.7" is turned into "3.6999999", and this number is so long that it spills into the part of the screen where the second number would normally be displayed. The result is to force the VIC to start a new line for the second number in the PRINT statement, and so to destroy the whole appearance of the table.

STR$ gives us much better control over the layout of decimal numbers. It takes a numerical argument, and produces a *string* of decimal digits, spaces, ., etc., which is the same as would have been displayed by the PRINT statement. The difference is that the result is *internal*, and can be manipulated and edited before being displayed. To illustrate this point, here is a short program which displays a number backwards:

```
10 INPUT "GIVE A NUMBER";X
20 X$=STR$(X)
30 FOR J=LEN(X$) TO 1 STEP -1
40 PRINT MID$(X$,J,1);
50 NEXT J
60 PRINT
70 GOTO 10
```

## ROUNDING

The first technique we shall examine is that of *rounding*. The VIC generally displays fractions to 8 decimal places, except that it leaves off trailing zeros. Usually 3 or 4 places are sufficient accuracy for your output. When a decimal is shortened by rounding, it is usual to add 1 to the last digit retained if the discarded portion starts with 5 or more. For instance, if the correct value of a number is 3.14159, the *rounded* value (to 3 places) is 3.142. On the other hand, the *rounded* value of 2.71828 is 2.718.

The mechanics of rounding are quite straightforward. To round a positive number, to 3 places, we *add* Ø.ØØØ5, and then *throw away* the fourth and subsequent places. These examples show the process at work:

| | |
|---|---|
| 3.14159 | 2.71818 |
| Ø.ØØØ5 + | Ø.ØØØ5 + |
| 3.14209 | 2.71878 |
| (discard Ø9) | (discard 78) |
| 3.142 | 2.718 |

It is clear that rounding will get rid of the truncation errors introduced by the VIC (since 3.69999999 rounded to 3 places is 3.7ØØ) and also avoid the embarrassing variations in the number of characters displayed.

The process of printing numbers rounded to 3 places is as follows:

(1) Add Ø.ØØØ5

(2) Using STR$ convert to a string form — say NN$

(3) Locate position of decimal point — say PP

(4) Display the left-hand part of NN$ up to 3 digits past the decimal point.

It could be that there is no decimal point in NN$. This would happen if the original value ended in ".9995" (such as "2.9995"). Then NN$ would appear as "3" with no decimal point. This has to be made a special case.

We can make this algorithm into a useful subroutine:

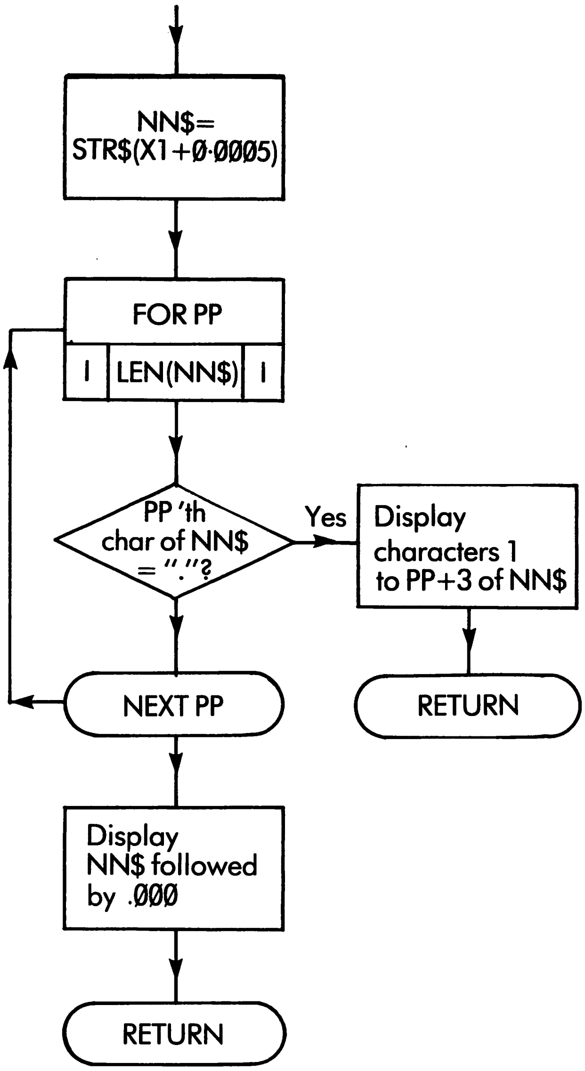
**Subroutine Specification**

*Purpose:* To display a positive number rounded to 3 decimal places.

*Lines:* 5ØØØ to 5Ø5Ø

*Input parameter:* X1 has value of number

*Local variables:* NN$, PP




The corresponding code is:

```
5ØØØ REM DISPLAY X1 TO 3 DECIMAL
       PLACES
5Ø1Ø NN$=STR$(X1+Ø.ØØØ5)
5Ø2Ø FOR PP=1 TO LEN(NN$)
5Ø3Ø IF MID$(NN$,PP,1)="."THEN PRINT
       LEFT$(NN$,PP+3);:RETURN
5Ø4Ø NEXT PP
5Ø5Ø PRINT NN$;".ØØØ";:RETURN:REM
       NO DECIMAL POINT IN NN$
```

A suitable 'driver' routine is a modified version of the program which gave us all the trouble originally:

```
1Ø FOR J=1 TO 7 STEP Ø.1
2Ø X1=J: GOSUB 5ØØØ
3Ø X1=J★J: GOSUB 5ØØØ
4Ø PRINT
5Ø NEXT J
6Ø STOP
```

If you run this program, you will see that all the difficulties disappear totally!

# EXPERIMENT 21·2

(a) Modify the Display subroutine discussed above so that the main program can select the number of decimal places used. This number will be supplied as a *parameter* in Y1. Hint: the constant to be added can be written as

Ø.5★1Ø↑−Y1

Test your subroutine out thoroughly and use it to display some new tables.

| Experiment 21.2 Completed | |
|---|---|

## AVOIDING WORD OVERFLOW ON THE SCREEN

The VIC is a superb machine in most respects, and excellent value for money; but even its designers will admit that it has a narrow screen. More expensive computers generally allow between 4Ø and 8Ø characters to appear on each line.

The narrow screen is not a serious drawback in programming, but it requires care to display messages so that the words don't spill over from one line to the next. If you are using a series of PRINT commands, you have to observe the following rules:

(1) No lines may be more than 22 characters long.

(2) If a line is exactly 22 characters, the PRINT command must follow the text with a semi-colon to prevent a blank line being forced. This, of course is because a character in the 22nd position always causes a new line to be started.



To end this unit, we'll describe a subroutine which automatically arranges text so as to avoid this problem.

Suppose we have a string of words separated by spaces. The string can be any length up to the maximum of 255 characters. If we simply PRINT it, it will be chopped up into 22-character lines without any regard to the positions of words and spaces. We have to devise a better method of dividing it into lines.

If the string is 22 characters or less, it can be displayed just as it is. Otherwise, we must examine the string and find the *largest* segment (starting at the beginning) which can be displayed without cutting a word in two. We display that segment, remove it from the front of the string, and start the process again on what is left. To find the largest segment, we look for a space starting at the 23rd character and searching *backwards*. To see why, consider the string

FRIENDS, ROMANS, COUNTRYMEN, LEND ME YOUR EARS

The 23rd character is an R; so we search backward until we come to the space at character 17. We display the *16*-character line,

FRIENDS, ROMANS,

and remove *17* characters from the front of the string, leaving

COUNTRYMEN, LEND ME YOUR EARS

The 23rd character is now a U. The next line to be displayed would be

COUNTRYMEN, LEND ME

and the final line since it is shorter than 22 characters, would just be

YOUR EARS

A subroutine specification, flow chart and code for this process are all given below.

**Subroutine Specification**

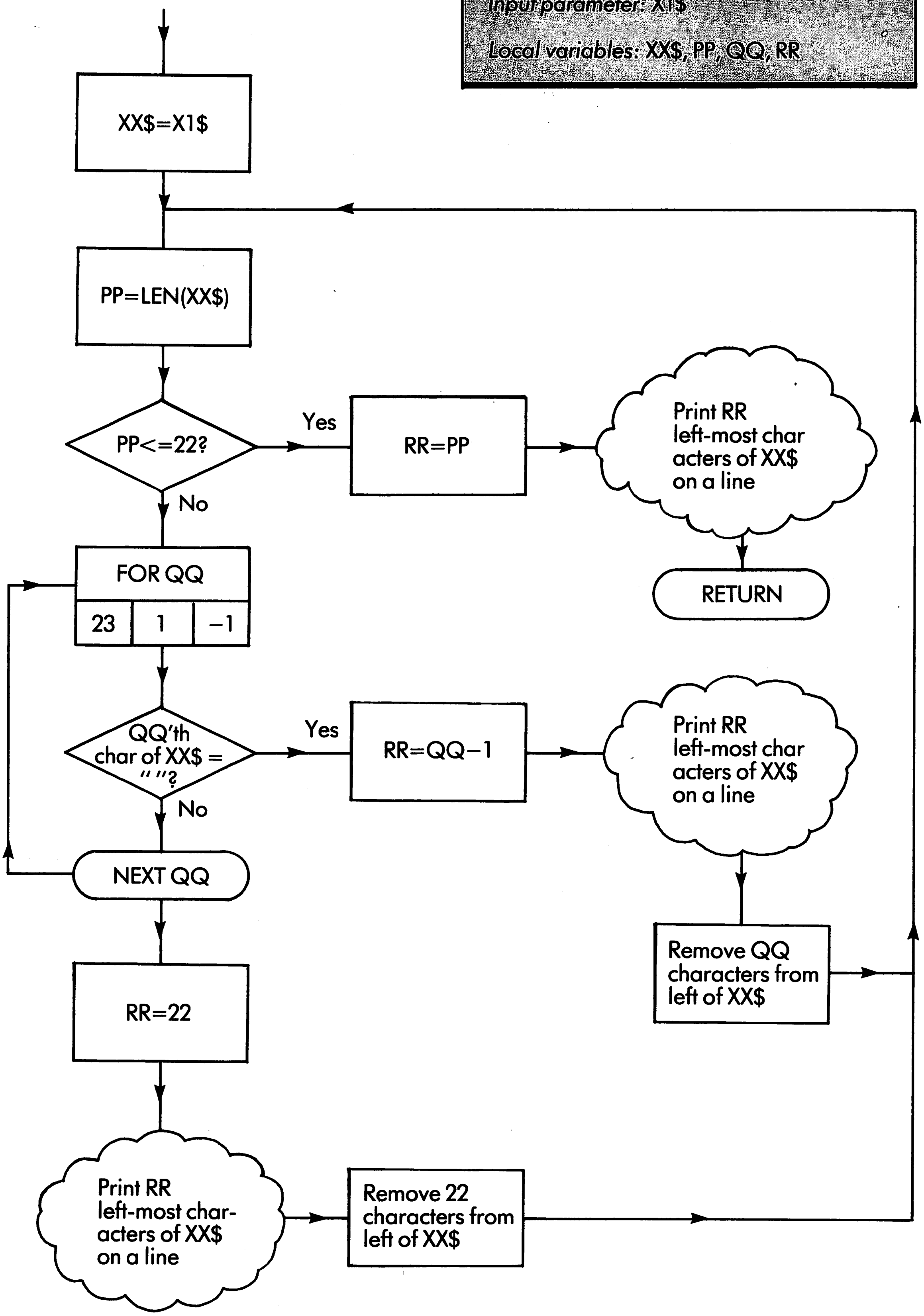
*Purpose:* To display string X1 so that words are not split across lines

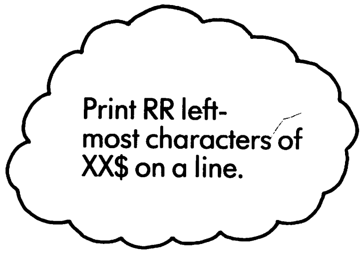
*Lines:* 5500-5600

*Input parameter:* X1$

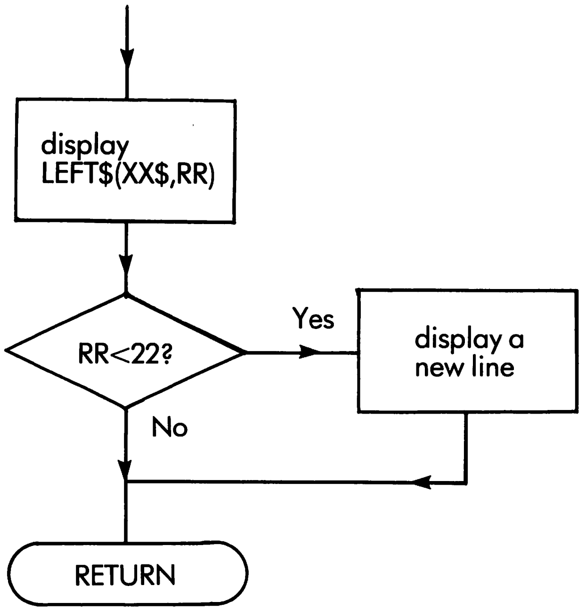
*Local variables:* XX$, PP, QQ, RR








(Internal subroutine).




```
5500 REM DISPLAY X1$ WITHOUT
     SPLITTING WORDS
5510 XX$=X1$
5520 PP=LEN(XX$)
5530 IF PP<=22 THEN RR=PP:GOSUB
     5580:RETURN
5540 FOR QQ=23 TO 1 STEP -1
5550 IF MID$(XX$,QQ,1)=" "THEN
     RR=QQ-1:GOSUB 5580:
     XX$=RIGHT$(XX$,PP-QQ):GOTO
     5520
5560 NEXT QQ
5570 RR=22: GOSUB 5580: XX$=RIGHT$
     (XX$,PP-22): GOTO 5520
5580 REM INTERNAL SUBROUTINE
5590 PRINT LEFT$(XX$,RR): IF RR<22 THEN
     PRINT
5600 RETURN
```

A suitable driver program for this subroutine is

```
10 X1$="TYPE ANY STRING
   AT ALL UP TO THREE LI
   NES LONG TO TRY OUT TH
   E LAYOUT SUBROUTINE"
20 GOSUB 5500
30 INPUT X1$: GOSUB 5500
40 GOTO 10
```

# EXPERIMENT 21·3



(a) The user of a program types a string which contains a number and a word, possibly (but not necessarily) separated by one or more spaces. The input string could be

```
   3 APPLES
or 174 PETS
or   1 QUEUE
```

Write a program which will extract the word and the number, and display them in the opposite order with the number doubled, thus:

```
APPLES    6
PETS    348
QUEUE     2
```

HINT: Use MID$ and VAL to extract the numbers.

(b) In reply to a question, a user can be expected to type a string like this:

```
I WANT 6 ORANGES 17 APPLES
2 PINEAPPLES 157 COCONUTS
AND 15 MELONS
```

Write a subroutine which analyses such a string and sets up variables as follows:

Array N1$: The *names* of the various items requested

Array Q1 : The quantities of the various items

Variable X: Number of different items requested.

For example, the sentence above should give:

```
N1$ (1) = "ORANGES"      Q1 (1) = 6
N1$ (2) = "APPLES"       Q1 (2) = 17
N1$ (3) = "PINEAPPLES"   Q1 (3) = 2
N1$ (4) = "COCONUTS"     Q1 (4) = 157
N1$ (5) = "MELONS"       Q1 (5) = 15

              X = 5
```

Your subroutine should ignore the words I, WANT, WOULD, LIKE, AND.

HINT: Scan along the string with a pointer, and use MID$ to extract sequences of letters or digits each terminated by a space.

| *Experiment 21.3 Completed* | |
|---|---|

The self test quiz for this unit is entitled UNIT21QUIZ.

# UNIT: 22

## MORE USES OF ARRAYS — Searching and Sorting:

This unit takes you further into the study of arrays and how they are used. We shall be looking at *searching* and *sorting*, two techniques which are vitally important to many modern computer applications.

The experiment at the end of Unit 20 asked you to write a program to search down a list of names held in an array. If there are only a few names this process is not difficult. You start at the top and work downwards stopping only when you find an exact match or "hit", or when you reach the bottom of the list and run out of names to search. A fragment of code involving such a search could be:



```
120 FOR J=1 TO 12
130 IF X$=A$(J) THEN 170
140 NEXT J
150 PRINT "NO MATCH"
160 STOP
170 PRINT "MATCH AT "; J
180 STOP
```

**Glossary**

X$: Name to be looked up
A$(1-12): Array of names to be searched
J: Pointer to current item in A$

In discussing methods of searching, the name (or number) being looked up is called the "target", and the act of matching it against an entry in the list is called a "comparison". In our example, the target is in X$, and the comparison occurs in line 130.

In practice, lists of names to be searched are often very much longer. The London telephone directory, for example, has about two million names. If your program had to search the whole of such a list from top to bottom, it would need to make two million comparisons. This would take a very long time, even at fast computer speeds.

Fortunately there are short cuts to the process. Suppose the names in your list are 'sorted' or arranged in increasing alphabetical order. You can use this fact in organising the search. For instance, you can begin by comparing the target with a name near the middle of the list. You may be very lucky and score a hit; but if you don't, one of two results is bound to happen:

(a) The target word is *less than* (i.e. nearer the beginning of the list) than the middle word.

or

(b) The target word is greater than (i.e. nearer the end of the list) than the middle word.

In the first case, you can be sure that if the target is in the list at all, it will be in the first half. Similarly, the second case tells you that the target can only be in the second half. Both ways, you have managed to eliminate half the list with two comparisons — one for equality, and one for relative order.

Once you know which half to use, you can apply the same process to that half, and identify a *quarter* of the original list, and then an eighth part, and so on.

Let's illustrate the process. Suppose the list of names is

ANDREW
ANTONIA
BEATRICE
CHRIS
FRANCES
HENRY
JIM
JOAN
JULIA
OLIVE
PETER
SUSAN
TIMOTHY
TOM
WILLIAM

We'll use *TOM* as a target word. To begin, we compare it with the middle name of the list, which is *JOAN*. Now TOM <> JOAN, so we don't score a direct hit. Furthermore TOM > JOAN, so we can eliminate all the list from JOAN upwards, and concentrate our search in the part from JULIA to the end.

The middle word of this part is *SUSAN*. TOM > SUSAN, so we again discard all the list except the bit between TIMOTHY and WILLIAM.

The middle word of the remaining section is TOM, which yields a direct hit.

If the target word isn't in the list at all, this quickly becomes obvious because the size of the list to be searched shrinks to nothing. For instance, take the target GEORGE:

stage 1: GEORGE < JOAN, so we use the list ANDREW—JIM (7 names)

stage 2: GEORGE > CHRIS, so we use the list FRANCES—JIM (3 names)

stage 3: GEORGE < HENRY, so we use the list FRANCES—FRANCES (1 name)

stage 4: GEORGE > FRANCES. No further subdivision possible, so GEORGE can't be in the list.

At this point, choose a few names, some in the list and some not, and go through the process of looking them up following the method we have just described.

## THE "BINARY CHOP"

If you think about it, you will see that at every stage the size of the list to be searched is roughly *halved*. It follows that if you double the size of the list, you'll add only one stage to the search process. As you move into larger lists, you begin to gain an overwhelming advantage over methods which rely on searching from top to bottom, looking at every name. The 'fast' method needs around 12 comparisons for a list with 1000 names, or around 21 for a list with a million! Since it relies on *cutting* a list in *two*, the method is called the "binary chop".

Let's code the method in BASIC. We assume that the list to be searched is 1ØØ items long and can be found in elements A$(1) to A$(1ØØ) of array A$. The target word is X$.

To organise the process, we must identify the part of the list in which the search is being conducted. To do this, we'll use two variables as *pointers*.

H 'points to' the top of the relevant part (that is, the element with the highest subscript)

L 'points to' the bottom of the relevant section (the element with the lowest subscript).

The phrase "points to" means "contains the subscript of". This is illustrated below:

A$(1) ANDREW
A$(2) ANTONIA
A$(3) BEATRICE — Variable L
A$(4) CHRIS — [• 5]
A$(5) FRANCES
A$(6) HENRY — Relevant section
A$(7) JIM
A$(8) JOAN — Variable H
.... — [• 7]

Thus the 'interesting' part of the list starts at A$(L) and ends at A$(H). If we ever find that H is *less than* L, the size of the list is zero and the search has failed.

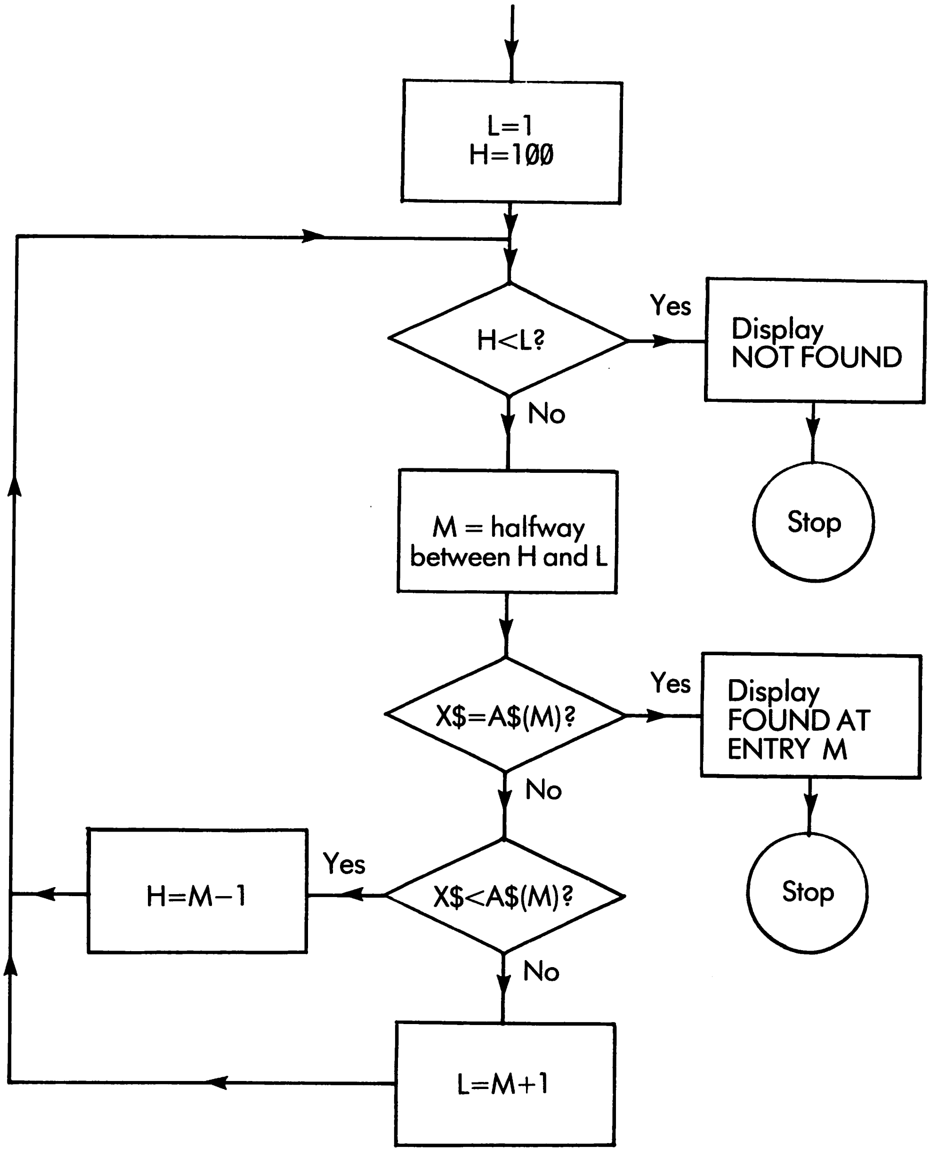
Finding the 'middle' word of the interesting part is quite easy. Its subscript is the 'average' of H and L, reduced to a whole number if necessary. The appropriate expression is

INT(Ø.5★(H+L)

It is convenient to assign this value to a variable M.

In planning the algorithm, we have to think carefully about changing the values of H and L. Suppose we find that the target word is greater than the middle word A$(M). This gives us a new *lower* limit of L=M+1, but doesn't change the *upper* limit H at all. Similarly, if the target word is less than A$(M), the new upper limit H will be M−1, but L won't need to be changed.

We can build these ideas into a flow chart:




**Glossary**

X$: Target word
A$(1-1ØØ): List of words to be searched. (In alphabetical order.)
L, H: Pointers to active part of list A$
M: Mid-point of active part of list

And a corresponding fragment of code could be

```
230 L=1 : H=100
240 IF H <L THEN PRINT X$; "NOT
      FOUND":STOP
250 M=INT(0.5*(H+L))
260 IF X$=A$(M) THEN PRINT X$; "FOUND
      AT ENTRY"; M:STOP
270 IF X$<A$(M) THEN H=M-1:GOTO
      240
280 L=M+1:GOTO 240
```

# EXPERIMENT 22·1

Turn the search code into a subroutine with the following specification:

**Subroutine Specification**

*Purpose:* To search an ordered list A$, between entries H1 and L1, for entry X$

*Lines:* 6ØØØ-61ØØ

*Parameters Input:* H1: Upper limit of search
L1: Lower limit of search
X$: Target word

*Output:* If a copy of X$ is found in A$, then M1 is its subscript. If a copy is not found, M1=1

Try out your subroutine with the following 'driver' program:

```
1Ø DATA BAIN, BEAVIS, BOWEY, BURNS,
   CLARK, FLEMING
2Ø DATA GORDON, GREEN, HOOD,
   KIDD, MACCABE, MALLY
3Ø DATA MARSHALL, MILLER, NORTH,
   PACK, PERKINS, REED, ROSE
4Ø DATA ROSS, SIMPSON, SMITH, SYKES,
   TEDFORD, WEBSTER, WOOD
5Ø DIM A$(26)
6Ø FOR J=1 TO 26 : READ A$(J) : NEXT J
7Ø INPUT "TYPE A NAME"; X$
8Ø L1=1 : H1=26 : GOSUB 6ØØØ
9Ø IF M1=-1 THEN PRINT X$; "NOT
   FOUND": GOTO 7Ø
1ØØ PRINT X$; " FOUND AT ENTRY"; M1
11Ø GOTO 7Ø
```

| *Experiment 22.1 Completed* | |
|---|---|

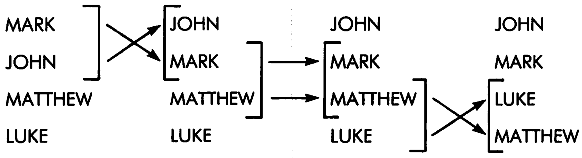
## THE BUBBLE SORT

In all the examples so far, the names were conveniently in alphabetical order when the program started. Suppose the names are supplied in random order? We have to arrange or 'sort' them ourselves.

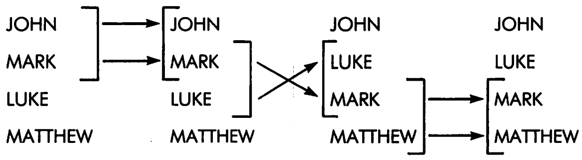
In a sorted list of names you can take any pair, and the one with the larger subscript will be alphabetically greater or equal to the one with the smaller subscript. This fact is the basis of the 'bubble sort', which is the simplest method of sorting.

We start with a list of names in random order. We 'sweep' through the list, and compare successive pairs of names (1 and 2, 2 and 3, and so on). If any pair is found to be out of order, these names are interchanged; each one is moved into the space previously occupied by the other one.

Here is an example of such a sweep:




This operation will always bring the greatest name to the bottom, but it won't necessarily leave the whole list in order. We have to 'sweep' again and again, until no more entries need to be changed. In this case the *second* sweep would give:




And the *third* sweep would give no interchanges, showing that the list was now in order.

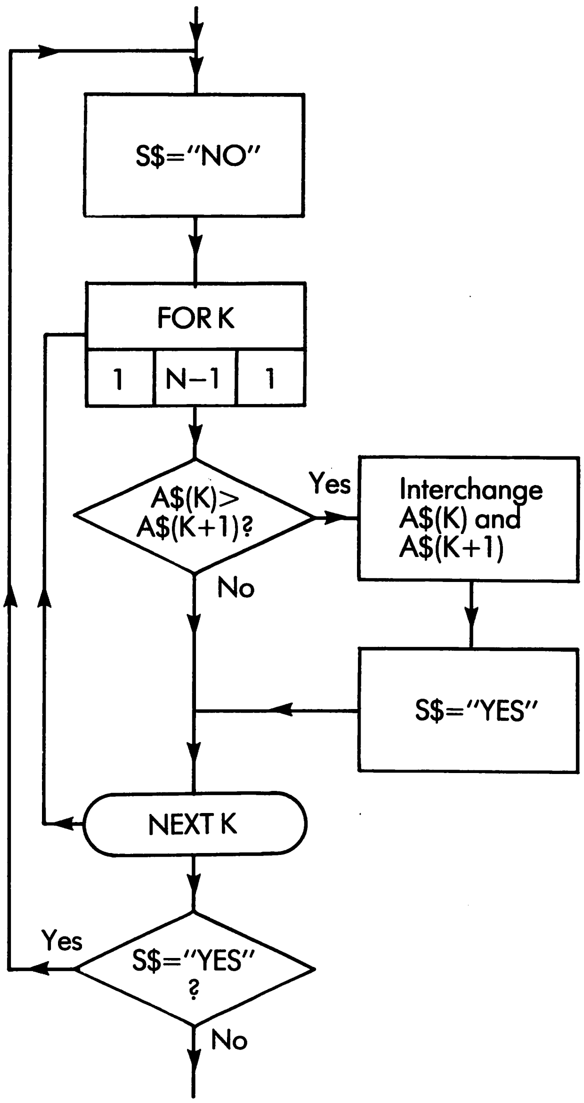
Interchanging two variables is not quite as simple as it seems. If you try to swap the values of X and Y by writing

X=Y : Y=X

it won't work; the first command destroys the initial value of X, and both variables end up with the original value of Y. We need a third, temporary variable — say — D, to hold the value of X until it is needed:

D=X : X=Y : Y=D

A flow chart for the bubble sort is as follows:




**Glossary**

S$: Marker for interchanges
A$(1−N): Array of words to be sorted
N: Number of words to be sorted
K: Pointer to A$

The corresponding code is

```
....
13Ø S$="NO"
14Ø FOR K=1 TO N−1
15Ø IF A$(K)>A$(K+1) THEN D$=A$(K):
      A$(K)=A$(K+1):A$(K+1)=D$:
      S$="YES"
16Ø NEXT K
17Ø IF S$ = "YES" THEN 13Ø
....
```

# EXPERIMENT 22·2

Turn the bubble sort into a subroutine with the following specification:

**Subroutine Specification**

*Purpose:* To sort items into alphabetical order

*Lines numbers:* 65ØØ-658Ø

*Parameters:* *Input:* List of items to be sorted in A1$(1) to A1$(N1)
*Output:* Sorted list appears in A1$(1) to A1$(N1)

*Local Variables:* KK, SS$, DD$

Try it out with your own data.

| *Experiment 22.2 Completed* | |
|---|---|

## QUICKSORT

The bubble sort is a fine simple method if the list is quite short (say 1Ø items or less) but as the list grows, so each sweep gets longer *and* you need more sweeps; so that the time needed for the job goes up as the *square* of the number of items to be sorted. This means that a list of 5Ø items will take about 25 times as long to be sorted as a list of 1Ø items.

There exists several sorting methods which are much faster than the bubble sort. One of them, called 'Quicksort' was invented by C. A. R. Hoare. In rough terms, the time it needs grows only as much as the number of items to be sorted.

Quicksort uses a programming technique called *recursion*, whereby a subroutine calls *itself* to do part of its job. Many people find the method hard to understand, particularly if it is expressed in BASIC which was not designed with recursive programs in mind. To use Quicksort effectively you don't have to understand it; nevertheless, here is a brief explanation which refers to the code given below.

The method starts with a list of items which are not in any special order. It takes the bottom one, calls it the 'key' element, and moves it into its correct final place in the list, making sure that all the items above it are also less, and all the items below are more. This is done by interchanging items if necessary. For example, the first stage in sorting a list of eight items is shown below:

| | | |
|---|---|---|
| 5 | 5 | |
| 18 | 4 | All items less than 12 |
| 23 | 6 | |
| 4 | 12 | Correct place for 12 (key) |
| 6 | 18 | |
| 17 | 23 | |
| 37 | 17 | All elements greater than 12 |
| 12 | 37 | |

The second stage consists of sorting all the items *above* the key element, and the third stage, of sorting the list of items *below* the key. For both these stages, the subroutine calls itself recursively, for sorting part of a list is basically the same problem as sorting the whole of one.

A good part of the subroutine below is concerned with providing the mechanism for the recursive calls. The array SS% and the pointer variable PP are used to remember exactly what is happening at any 'level' of control so that all the calls and returns are made in an orderly manner. Command 617Ø is equivalent to a RETURN.

In the subroutine, lines 6Ø4Ø to 6Ø9Ø carry out stage 1; 61ØØ to 613Ø is concerned with stage 2 (which can be skipped if the 'list' above the key element is less than two items long). Lines 614Ø to 616Ø look after stage 3, and lines 6Ø1Ø, 6Ø2Ø, 611Ø, 613Ø, 615Ø and 617Ø are all needed for recursion. The subroutine includes two unfamiliar aspects: an array name ending with % and a command with the keyword ON. Both will be discussed later.

**Subroutine Specification**

*Purpose:* To sort items into numerical order, using Hoare's Quicksort algorithm.

*Line numbers:* 6ØØØ-618Ø

*Parameters:* *Input* : List of numbers to be sorted in A1(1) to A1(N1). Number of items in N1
*Output:* Sorted list appears in A1(1) to A1(N)

*Local Variables:* SS, SS%, AA, BB, XX, YY, ZZ, DD, PP

*NOTE:* (i) SS% must not be used *anywhere else* in the program if the sort subroutine is called more than once.
(ii) The subroutine may be used to sort *strings* instead of numbers if the following substitutions are made throughout:

A1$ for A1; ZZ$ for ZZ; DD$ for DD

```
6ØØØ REM QUICKSORT : SORTS N1
     ELEMENTS OF A1
6Ø1Ø IF SS=1 THEN 6Ø3Ø
6Ø2Ø DIM SS%(N1) : SS=1 : REM DECLARE
     STACK
6Ø3Ø AA=1 : BB=N1: SS%(Ø)=1:PP=1
6Ø4Ø XX=AA: YY=BB: ZZ=A1(BB)
6Ø5Ø IF XX>=YY THEN 6Ø9Ø
6Ø6Ø IF A1(XX)<=ZZ THEN XX=XX+1:
     GOTO 6Ø5Ø
6Ø7Ø IF A1(YY)>=ZZ THEN YY=YY-1 :
     GOTO 6Ø5Ø
6Ø8Ø DD=A1(YY): A1(YY)=A1(XX):
     A1(XX)=DD: GOTO 6Ø5Ø
6Ø9Ø A1(BB)=A1(XX): A1(XX)=ZZ
61ØØ IF XX-AA<= 1 THEN 614Ø
611Ø SS%(PP)=XX: SS%(PP+1)=BB:
     SS%(PP+2)=2: PP=PP+3
612Ø BB=XX-1: GOTO 6Ø4Ø
613Ø PP=PP-3: XX=SS%(PP):
     BB=SS%(PP+1)
```

```
6140 IF BB-XX<=1 THEN 6170
6150 SS%(PP)=3: PP=PP+1: AA=XX+1:
     GOTO 6040
6160 PP=PP-1
6170 ON SS%(PP-1) GOTO 6180,6130,
     6160
6180 RETURN
```

This complete subroutine entitled "QUICKSORT" can be found on the cassette tape.

## SORTING TIMES COMPARED



Quicksort is so much more complicated than the Bubble sort that you may wonder if it is worth the trouble of using? You can judge for youself from this table which shows the times needed to sort arrays of various sizes. The figures were found by running both types of sort on a VIC and timing them.

| *Size of Array* | *Time (Quicksort)* | *Time (Bubble sort)* |
|---|---|---|
| 20 | 2 | 5 |
| 40 | 5 | 22 |
| 60 | 8 | 47 |
| 80 | 14 | 93 |
| 100 | 17 | 138 |
| 120 | 20 | 192 |
| 140 | 24 | 282 |
| 160 | 27 | 357 |
| 180 | 31 | 445 |
| 200 | 37 | 569 |

## THE VIC MEMORY CAPACITY

When you start using arrays, you are soon likely to come up against the problem of space in the VIC store. This is because arrays gobble up a great deal of space very quickly; each element of a number array uses up 5 bytes of store, and every string element uses 3 bytes, *plus* the space needed for the string itself. There are also small additional overheads for each array.

In this section, we shall look at the way the VIC store is organised. A useful tell-tale is the built-in function FRE(0) which tells you how many bytes remain unused at any moment. When you first switch the VIC on, the message comes up:

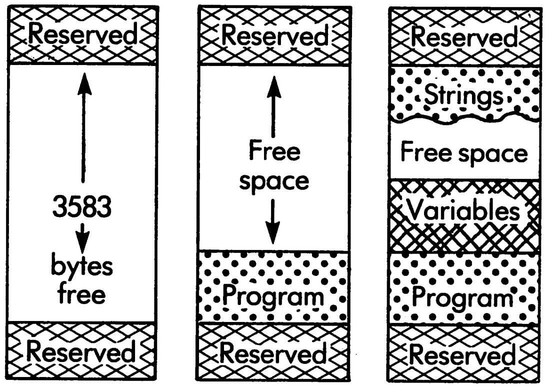
3583 BYTES FREE (More if you have a RAM-pack fitted)

If you now type

PRINT FRE(0)

the machine replies 3581, because 2 bytes are used up in obeying the FRE function itself.

The overall situation is shown in Part (i) of the diagram below; of the 5120 bytes in the VIC, 1537 are reserved for various purposes and the rest are still free.




(i) (ii) (iii)

Next, you might type in a program or load one from a cassette tape. The program is put away in the bottom of the free section of store, taking up roughly one byte for each character. The result is shown in (ii) of the diagram.

Now you start the program. The machine begins to obey your commands; and as soon as it comes across any variable referred to for the first time, it allocates the necessary storage space in the area immediately adjacent to the program itself. DIM commands are given space in the same area, and can use it up very quickly; an innocent-seeming command like

DIM A(200)

will cost over 1000 bytes.

Once space for a variable or array has been allocated, it can't be clawed back and used for any other purpose until the program is stopped.

Strings are managed differently. Strings that are read directly from DATA statements in the program take up no extra space at all. Strings which are read from the keyboard or constructed with "+", MID$ and other string functions are placed at the *other* end of the store, leaving free space between the strings and the variables. This is illustrated in Part iii of the diagram. The space used by strings *is* recoverable; when a string is no longer needed it can be discarded and the space is returned to the free area.

(If you think this is a complicated process, you are right — it is called "garbage collection". Fortunately it is completely automatic and you don't need to know anything about it.)

Apart from the overall size limit, the VIC store isn't partitioned in any way. You can have as much program, variables and strings as you like provided that the total doesn't exceed the free space available.

Key in and run the following programs, and think about its results in the light of this discussion:

```
10 PRINT "FREE SPACE", "AFTER LINE"
20 PRINT FRE(0), 10
30 X=0
40 PRINT FRE(0), 30
50 DIM A(20)
```

```
 6Ø PRINT FRE(Ø), 5Ø
 7Ø DIM N$(5)
 8Ø PRINT FRE(Ø),7Ø
 9Ø C$="A STRING"
1ØØ PRINT FRE(Ø), 9Ø
11Ø D$ = "ANOTHER "+" STRING"
12Ø PRINT FRE(Ø), 11Ø
13Ø D$=" "
14Ø PRINT FRE(Ø), 13Ø
15Ø C$=" "
16Ø PRINT FRE (Ø), 15Ø
17Ø STOP
```

## EXHAUSTING THE VIC'S MEMORY SPACE

We can now explain the various ways you can run out of space.

(a) If your program is too long, you will hit the space limit before you finish entering it. This is most likely to happen if you try to load a program which was developed on a VIC with a bigger store.

(b) If you have too many variables, or your arrays are too long, you will get an

OUT OF MEMORY

error when the machine tries to allocate a new variable or obey a DIM command.

(c) If you try to produce and store too many strings at the same time, you will also get an OUT OF MEMORY message.

Running out of store is always a frustrating experience. You get the feeling that if only your machine were a little bigger your program would work perfectly. Here are some good ways of overcoming the difficulty:

1. Reduce the size of your arrays to the minimum. Don't guess the number of items the user is going to supply — make the program find out and dimension its arrays accordingly. For instance, it is generally better to write:

```
1Ø INPUT N
2Ø DIM X(N),Y$(N)
```

than

```
1Ø DIM X(1ØØ), Y$(1ØØ)
2Ø INPUT N
```

2. If you have arrays with numbers, and you know — for certain — that every element is bound to be a whole number in the range — −32767 to +32767, you can use *integer arrays*. Integer arrays have names which end in % (like A1% and JJ%) and they use up only two bytes per element, rather than the usual five. To modify a program to use integer arrays, you would change — say —

```
1Ø DIM N(5ØØ)
```

into 1Ø DIM N%(5ØØ)

and then every mention of an array element such as N(J) into N%(J).

3. As your program runs, make it discard all the strings it doesn't need. To do this, you assign the null string to the appropriate variable:

```
X$=M$+"MARRIED" + "F$"
......
...... (X$ no longer needed)
X$=" "
```

4. Examine your program carefully and see if you can find an algorithm which needs less space than the one you are using. Do you really need to store all the elements of an array, or can you perhaps calculate and use them one by one?

5. If you use a long number or string constant in several places, assign it to a variable and then use the name of the variable instead. For instance, consider:

```
 9Ø PRINT "RESULT IS CONFIRMED AS";
1ØØ PRINT X/2.718281828
11Ø GOTO 15Ø
12Ø Y=Z/2.718281828
13Ø Q=Y+2.718281818
14Ø PRINT "RESULT IS CONFIRMED AS";
    Q+2
15Ø .......
```

This program will take *slightly* less space if you write it as:

```
 1Ø L$="RESULT IS CONFIRMED AS":
    E=2.718281828
.....
 9Ø PRINT L$;
1ØØ PRINT X/E
11Ø GOTO 15Ø
12Ø Y=Z/E
13Ø Q=Y+E
14Ø PRINT L$; Q+2
15Ø .....
```

Remember that the character $\pi$ (one byte) stands for 3.14159165..... so you can prove this by typing PRINT $\pi$.

6. Buy extra memory cartridges from your Commodore dealer. These are available as plug in cartridges giving an extra 3K, 8K or 16K RAM.

If you are really desperate for space, here are some other things you can do to shoehorn your program into a limited store. They are not really recommended; they make your programs more difficult to understand and may introduce errors.

1. Remove messages displayed on the screen, and rely on written explanations.
2. See if you can share variables for more than one purpose.
3. Go through your program and remove any spaces (except in strings). For instance, you will save 5 bytes by changing

   IF A<5 AND B>7 THEN 4ØØ

   into IFA<5ANDB>7THEN4ØØ

   You will also save a little space by putting as many commands as you can into each line.
4. As a final gesture, take the REM's out of your program. You will be like Jules Verne's hero Phineas Fogg. To get to Liverpool on time to win his bet he ripped out the deck of his ship and burnt it in the boiler. He won but ruined the ship in the process!

## TWO DIMENSIONAL ARRAYS

As you can see, arrays are useful in problems where the program has to handle many different variables of the same type. In some problems it is natural to arrange these variables in a square or rectangular table rather than a simple ordered list. Consider a chess-playing program. It must 'know' what piece — if any — occupies each square of the board. Every square can be repeated by a variable whose value reflects the piece in that square. The 64 variables are arranged in a table with 8 rows and 8 columns, which model the shape of the board.

BASIC allows for two-dimensional (and even three-dimensional or more) arrays. A typical declaration of a two-dimensional array would read,

DIM X(5,7)

This command sets up an array called X, in which every element is a number. The array has (5+1) or 6 rows and (7+1) or 8 columns: 48 elements in all. A picture of it is:

| | Ø | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Ø | X(Ø,Ø) | X(Ø,1) | X(Ø,2) | X(Ø,3) | X(Ø,4) | X(Ø,5) | X(Ø,6) | X(Ø,7) |
| 1 | X(1,Ø) | X(1,1) | X(1,2) | X(1,3) | X(1,4) | X(1,5) | X(1,6) | X(1,7) |
| 2 | X(2,Ø) | X(2,1) | X(2,2) | X(2,3) | X(2,4) | X(2,5) | X(2,6) | X(2,7) |
| 3 | X(3,Ø) | X(3,1) | X(3,2) | X(3,3) | X(3,4) | X(3,5) | X(3,6) | X(3,7) |
| 4 | X(4,Ø) | X(4,1) | X(4,2) | X(4,3) | X(4,4) | X(4,5) | X(4,6) | X(4,7) |
| 5 | X(5,Ø) | X(5,1) | X(5,2) | X(5,3) | X(5,4) | X(5,5) | X(5,6) | X(5,7) |

Each element in the array has *two* subscripts: a row number and column number. For example, X(3,4) is in row 3 and column 4. Apart from this fundamental difference, everything you know about one-dimensional arrays can be extended to two dimensions.

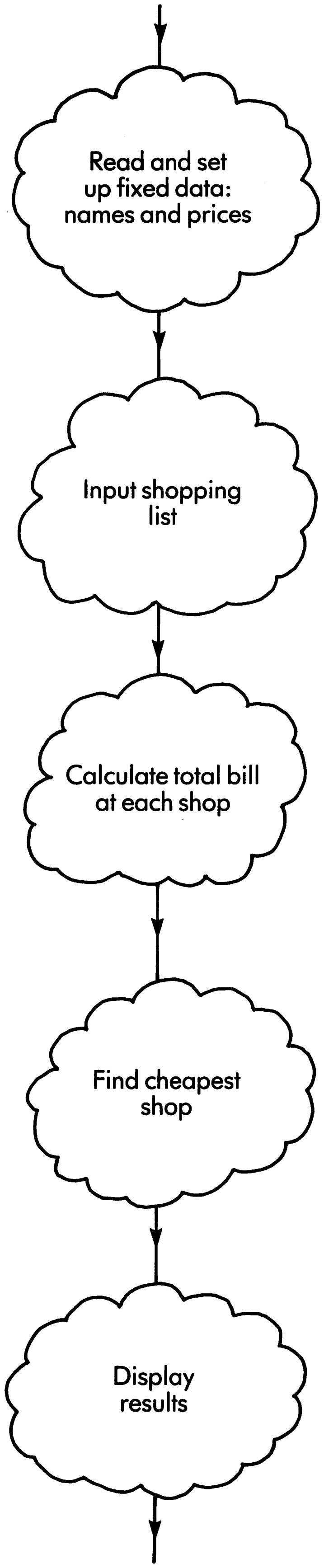
We move straight on to an illustration. Suppose you have done a survey and discovered the price of some basic commodities in each of five shops in your district. You might express the results in a table like this:

| | FINE FARE | ASDA | SAINS-BURYS | CO-OP | FRASERS |
|---|---|---|---|---|---|
| FLOUR | 29 | 31 | 27 | 26 | 32 |
| POTATOES | 15 | 12 | 13 | 24 | 33 |
| BUTTER | 47 | 49 | 40 | 45 | 39 |
| SUGAR | 22 | 20 | 19 | 27 | 29 |
| CHEESE | 94 | 80 | 103 | 107 | 99 |
| APPLES | 32 | 18 | 22 | 27 | 21 |
| | (All prices in pence per pound) | | | | |

The problem to be solved is, given a particular shopping list, which is the cheapest shop to visit? A 'user's' view of the program could be:

```
PLEASE STATE YOUR NEEDS
IN LBS.

FLOUR?      3
POTATOES?  14
BUTTER?     1     Typed by user
SUGAR?      Ø
CHEESE?     2
APPLES?     3

BEST VISIT ASDA
WHERE YOU PAY
3 LBS FLOUR     :  93
14 LBS POTATOES: 168
1 LB BUTTER     :  49
Ø LBS SUGAR     :   Ø
2 LBS CHEESE    : 16Ø
3 LBS APPLES    :  54

    TOTAL       : 524p
```

The basic algorithm is straightforward:




We begin by selecting some variables and their names. We'll certainly need to store the names of the shops and the various articles of food. Suitable variables are:

F$(1) to F$(6) for the foods

and S$(1) to S$(5) for the shops

Next we need arrays to show the quantity of each food needed and the corresponding amount paid (or total) at each shop. Suitable variables are:

F(1) to F(6) for quantities, and T(1) to T(5) for totals.

Finally, we'll use a two-dimensional array to hold the price table. A declaration such as:

DIM P(6,5) will do well.

Note that we consistently ignore elements with subscripts of Ø. This is common in small problems.

The actual code is quite straightforward, if a little lengthy. It is:

```
 1Ø DATA FLOUR, POTATOES, BUTTER,
    SUGAR, CHEESE, APPLES
 2Ø DATA FINE FARE, ASDA, SAINSBURYS,
    COOP, FRASERS
 3Ø DATA 29, 31, 27, 26, 32
 4Ø DATA 15, 12, 13, 24, 33
 5Ø DATA 47, 49, 4Ø, 45, 39
 6Ø DATA 22, 2Ø, 19, 27, 29
 7Ø DATA 94, 8Ø, 1Ø3, 1Ø7, 99
 8Ø DATA 32, 18, 22, 27, 21
 9Ø DIM F$(6), S$(5), F(6), T(5), P(6,5)
1ØØ FOR K=1 TO 6 : READ F$(K) : NEXT K
11Ø FOR J=1 TO 5: READ S$(J): NEXT J
12Ø FOR K=1 TO 6: FOR J=1 TO 5
13Ø READ P(K,J)
14Ø NEXT J,K
15Ø PRINT " [SHIFT] and [CLR HOME] PLEASE
    STATE YOUR"
16Ø PRINT "NEEDS IN LBS"
17Ø FOR K=1 TO 6
18Ø PRINT F$(K);: INPUT F(K)
19Ø NEXT K
2ØØ FOR J=1 TO 5
21Ø FOR K=1 TO 6
22Ø T(J)=T(J)+F(K)★P(K,J)
23Ø NEXT K,J
24Ø M=T(1): N=1
25Ø FOR J=2 TO 5
26Ø IF T(J)<M THEN M=T(J): N=J
27Ø NEXT J
28Ø PRINT "BEST VISIT"; S$(N)
29Ø PRINT "WHERE YOU'LL PAY" : PRINT
3ØØ FOR K=1 TO 6
31Ø PRINT F(K); "LB . ";
32Ø IF F(K)<> 1 THEN PRINT " [SHIFT]
    and [⇐ CRSR ⇒] [SHIFT] and [⇐ CRSR ⇒] S. ";
33Ø PRINT F$(K); TAB(15); F(K)★P(K,N)
34Ø NEXT K
35Ø PRINT: PRINT"TOTAL ="; TAB(15); M;
    "P. "
36Ø STOP
```

One or two minor points must be clarified.

(a) All the arrays are declared together in one command. This is shorter than writing

```
 90 DIM F$(6)
100 DIM S$(5)
....
```

and so on.

The limit to the number of arrays which may be declared is set by the maximum line length of 88 characters.

(b) The sequence of commands

```
NEXT J
NEXT K
```

can be compressed into

```
NEXT J,K
```

This applies equally well to any control variables, and to any number of them (although more than two is rare).

(c) The phrase "TAB (15)" in command 35Ø makes the machine move its internal cursor to the 15th column in the screen (if the cursor is not already there). It is used here to line up the amount paid for each article of food.

In general, the brackets may contain any expression. Thus the program

```
1Ø FOR J=1 TO 2Ø
2Ø PRINT TAB(J); "/"
3Ø NEXT J
4Ø STOP
```

will display a sloping line across the screen.

In the program on the previous page, commands 31Ø-32Ø are partly concerned with displaying "LB . " for one pound, or "LBS . " if more (or less) than one. 31Ø displays "LB . ", and 32Ø moves the cursor back and puts in the "S" if necessary.

Note that J is consistently used for the shop number, and K for the food type number. P(K,J) is therefore the price of food number K at shop number J.



# EXPERIMENT 22·3

This experiment is in two parts:

(a) A large class of students have a competitive exam. The teacher produces a set of marks like this:

| | |
|---|---|
| ADAMS | 27 |
| BRIGGS | 66 |
| CHILVERS | 89 |
| DALE | 38 |
| ..... | |

and so on.

The rules say that only the top 25% (one quarter) of the students may pass.

Write a program which can read in the original mark list and display the names of the students who pass. Assume that there are not more than 1ØØ students, and that the mark of the last student is followed by the dummy name "ZZZZ".

HINT: Sort a copy of the marks using the QUICKSORT subroutine on the cassette tape, and find the 'minimum' pass-mark a quarter way down the sorted list. Use it to pick out the students who pass.

(b) The game of 'life' was invented by R. Conway, an English Mathematician. It concerns the life history of a colony of bugs which live in a rectangular area, one to each cell.

The colony lives from generation to generation. The fate of each bug is determined by the following rules:

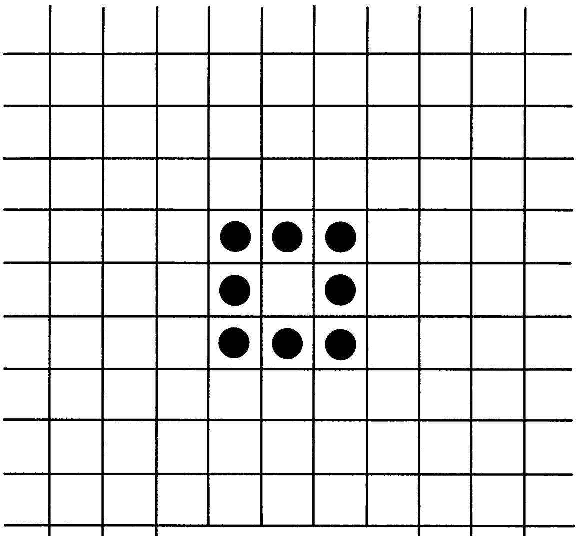
1. If a bug has 1 or fewer immediate neighbours, it dies of loneliness.

2. If it has 4 or more neighbours, it dies of overcrowding.

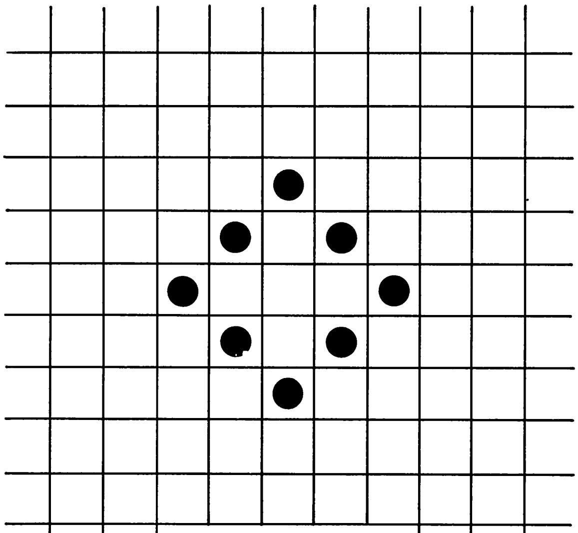
3. If it has 2 or 3 neighbours it survives to the next generation.

Furthermore, if an empty cell has exactly 3 bugs in neighbouring cells, a new bug is born in that cell.

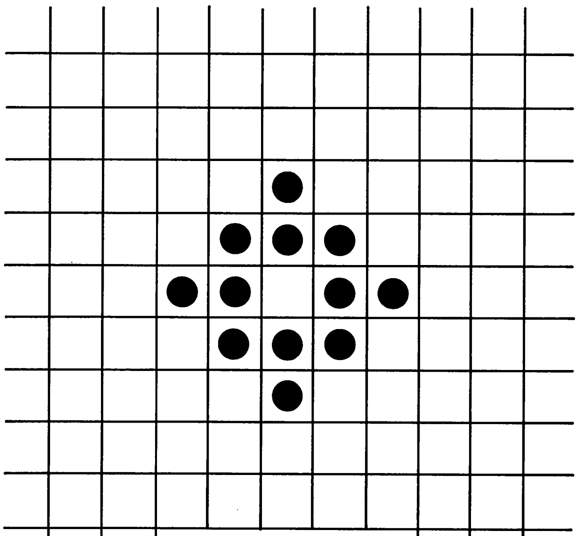
To give an example, consider



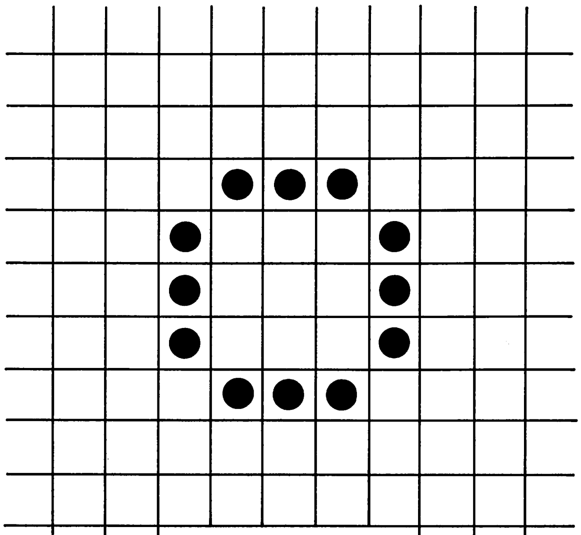
*Start*



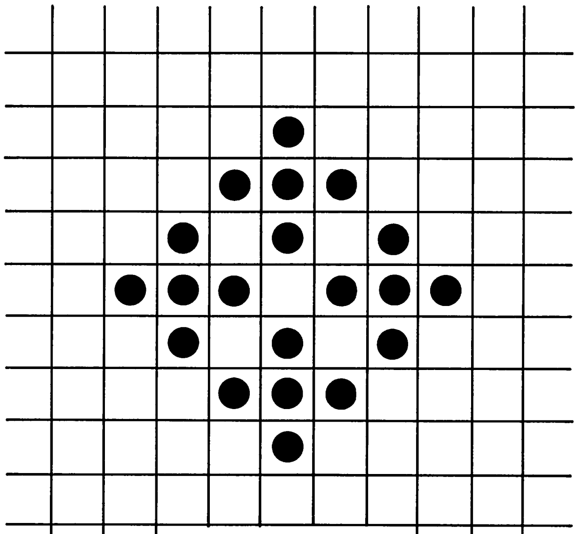
*2nd generation*



*3rd generation*



*4th generation*



*5th generation*



Write a program to play the 'life' game on a 9 x 9 array. Each element should be a string holding one character (say "★" or space). The program should read in a 'starting position' and then display successive generations until it is stopped.

This is a challenging experiment, and you may use the partial program "LIFESTART" on the cassette tape. The program reads a starting position into array X$(9,9) (lines 1Ø-18Ø) and then displays it (lines 19Ø-26Ø). The partial program also declares an array Y$(9,9), which you will find useful in going from one generation to the next.

HINT (1): To determine a 'next generation', use the array Y$, which has already been declared for you. When you have built up the complete generation in Y$, copy it back into X$.

HINT (2): If you are looking at cell X$(J,K) (where J and K are subscripts) the 8 neighbouring cells will be:

X$(J−1,K−1), X$(J−1,K) X$(J−1,K+1)

X$(J, K−1) X$(J, K+1)

X$(J+1,K−1), X$(J+1,K), X$(J+1,K+1)

To prevent references to cells which aren't in the array at all, make the assumption that the border cells are always empty, and confine operations to the 7 'internal' rows and columns.



Now check your answer against the one given in Appendix C.

| *Experiment 22.3 Completed* | |
|---|---|

# UNIT: 23

## A CLOSER LOOK INSIDE THE VIC

A computer is an extremely complicated device. If you try to explain everything about it to a beginner, all at once, you'll leave him bewildered and hopelessly confused long before he can do anything interesting or useful. Instead, you can treat the machine like a parcel at a kid's party: something with lots of paper wrappings which you can strip off one layer at a time. If you conceal unnecessary detail, you can always arrange for the outermost layer to look quite simple. For example, many people will always think of the VIC just as a machine which plays games which come on cassette or plug-in cartridges. For people who don't want to know about programming, this is a perfectly reasonable and useful level of understanding.



Some people like to dig deeper. You, the reader, are already aware of the VIC as a machine which stores and obeys BASIC programs. This again is a useful and important level of understanding, because it lets you use the machine in all sorts of original and interesting ways; but it leaves out detail about how information is stored, how the machine obeys a program, and how it actually works.

In this Unit we'll have to go one layer further towards the innermost mechanism of the VIC. You'll find that the description of the VIC's memory seems different from the picture presented in previous units. This is because we are seeing the memory from a new and closer viewpoint. Both descriptions are true and each is appropriate to the level at which the system is being described.

This Unit explores the mysterious PEEK and POKE commands. We have to begin with two warnings:

1. Unlike the rest of the book, the material in this Unit applies *only* to the VIC and can't be used with any other computer. Most personal computers support PEEK and POKE commands, but they do different things on different machines!

2. PEEK and POKE are sneaky commands which let you further into the inner workings of the VIC than other BASIC commands. This means that a level of protection is by-passed. A program with errors can corrupt the VIC's software and make it behave in a very strange manner. For instance, the keyboard might become totally dead, or reams of rubbish might be displayed on the screen. Again, the computer could refuse to obey simple commands like 'LIST' or 'RUN'. If this happens, you can *always* regain control by switching off the machine for 30 seconds. Since this deletes your program, it is doubly important to frequently store your program on a cassette if you are using many PEEK's and POKE's.

   Software corruption is a temporary effect. It is absolutely impossible to do your VIC any permanent damage by running any program, no matter how full of mistakes it may be.

To understand PEEK and POKE, we must first learn a little more about the VIC computer itself.

A diagram of VIC (at the level appropriate to this chapter) is shown on the next page.

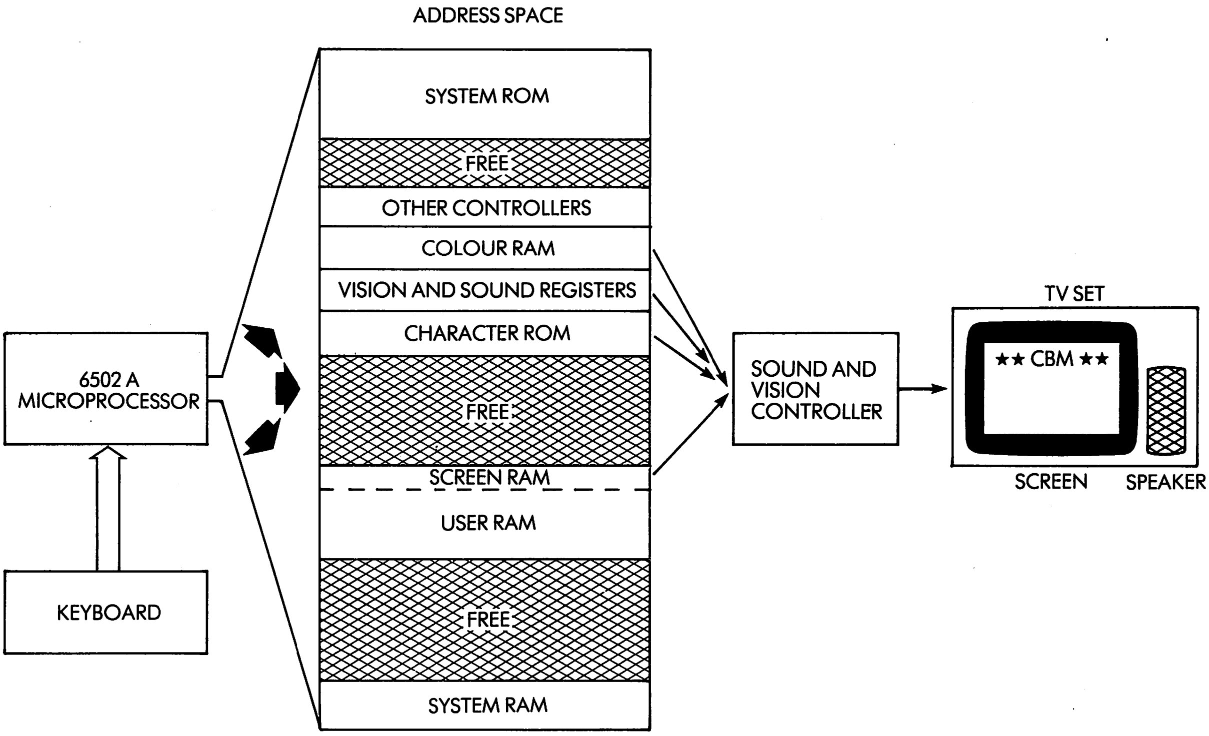





*Figure 23.1*

The machine consists of several parts:

(a) An "address space" which holds modules of memory of different kinds.

(b) A microprocessor, which takes *instructions* out of the memory, and executes them. Most of the instructions result in changes being made to the contents of the memory. The instructions are similar to, but simpler than the commands in a BASIC program.

(c) A keyboard, which is looked after by the microprocessor.

(d) A 'sound and vision controller', which produces the picture and the sound-effects you see on the TV set.

The address space is like a rack into which you can plug separate 'segments' of memory. Altogether there is room for 65536 bytes of store, and the slots are numbered from Ø up to 65535. Most memory segments hold 1Ø24 bytes (or multiples of 1Ø24, such as 2Ø48 or 4Ø96 or 8192), so the number 1Ø24 is called a 'Kilo', or "K" for short. The full capacity of the address space is exactly 64 Kilobytes.

The address space is only partially filled, and the memory segments it contains are of three different kinds:

(a) *RAM* stands for "Random Access Memory". The contents of each byte can be read and altered as often as necessary.

(b) *ROM* is "Read-Only Memory". The contents of each byte is fixed for ever when the ROM is manufactured. Once the ROM is in the machine, the VIC can read the byte but can't alter it.

(c) Registers and controllers are special devices which have particular jobs such as helping to produce sounds and pictures. They also appear to be like memory, so that the micro-processor can read and change their contents.

## THE VIC'S MEMORY ORGANISATION:

When you buy a basic VIC, you will find the address space inhabited as follows:

Addresses Ø-1Ø23: 1K of RAM reserved by the VIC for its own use. It is here that the machine keeps track of the time, a record of the position of the cursor, and so on.

Addresses 1Ø24-4Ø95: Unoccupied. (This is where the 3K RAM cartridge resides.)

Addresses 4Ø96-8191: 4K of "User RAM" which serves many purposes: it holds your BASIC program, your data (variables and strings) and 5Ø6 bytes near the top are used to control the TV screen in a way to be described later.

Addresses 8192-32767: Free. (This is where the 8K or 16K RAM plug in cartridges reside.)

Addresses 32768-36863: 4K of ROM, which holds a description of the 'shape' of every character which can be displayed on the screen. We shall be going into more detail later on.

Addresses 36864-37887: This area holds the vision and sound registers, and some other controllers we needn't describe.

Addresses 37888-38912: 1K of RAM in which 5Ø6 words are used to control the colour of each character of the screen.

Addresses 38912-49151: Empty.



Addresses 49152-65535: 16K of ROM which holds the VIC's own software: programs which organise screen editing, program loading and saving, and the execution of users' programs.

As you can see, there is a substantial amount of free space — 37K in all. This space can be filled up, either by RAM modules which increase the amount of user RAM, or with ROM modules which contain ready-made programs. For instance, the VIC 1210 3K RAM pack which plugs into the back of the machine slots straight into the 3K gap between 1Ø24 and 4Ø95.

## WHAT IS A BYTE?

Now we'll look at the way every individual byte is arranged. It consists of 8 separate elements called 'bits', each of which can have only two possible values. There are 256 possible combinations of bits in a byte. If we call the values of a bit Ø and 1, some of the combinations are:

ØØØØØØØØ   Ø1Ø111ØØ   1Ø1Ø1Ø11

The *meaning* of a byte depends entirely on the context in which it is used. In the VIC it could be

(a) A code for a specific character (e.g. "A" could be ØØØØØØØ1).

(b) A pattern of up to 8 dots on the screen.

(c) An instruction for the microprocessor to carry out some action.

(d) A number, where the bits are interpreted by a mathematical rule called the 'binary system'.

(e) A byte can also be one of a group of 5 bytes which make up a number variable, or one of several in a string variable.

To appreciate bytes it helps to understand the binary system. Fortunately it isn't very hard. An 8-bit byte can be converted into an ordinary number by the following rule:

The leftmost bit scores 128 if 1, zero if it is Ø
The next bit scores 64 if 1, zero if it is Ø
The next bit scores 32 if 1, zero if it is Ø
The next bit scores 16 if 1, zero if it is Ø
The next bit scores 8 if 1, zero if it is Ø
The next bit scores 4 if 1, zero if it is Ø
The next bit scores 2 if 1, zero if it is Ø
The rightmost bit scores 1 if 1, zero if it is Ø.

To give an example, let's take the byte 1Ø11Ø11Ø.

The bits score 128, 32, 16, 4 and 2, so the corresponding number is 128+32+16+4+2 or *182*.

Sometimes you have to convert a number to its byte equivalent. The number must be less than 256 (and not less than Ø), for the conversion to work. The method is:

1. If you can subtract 128, do so and write down a 1. Otherwise write down a 'Ø'.

2. If you can subtract 64, do so and write down a '1' to the right of the previous symbol. Otherwise write down a 'Ø'.

3-8. Do the same for 32, 16, 8, 4, 2 and 1.

As an example, take the number 2Ø1. The process gives:

| | |
|---|---|
| (1) 2Ø1−128 = 73, so put down | 1 |
| (2) 73−64 = 9, so put down | 11 |
| (3) 9−32 won't go, so put down | 11Ø |
| (4) 9−16 won't go, so put down | 11ØØ |
| (5) 9−8 = 1, so put down | 11ØØ1 |
| (6) 1−4 won't go, so put down | 11ØØ1Ø |
| (7) 1−2 won't go, so put down | 11ØØ1ØØ |
| (8) 1−1 = Ø, so put down | 11ØØ1ØØ1 |

## THE 'PEEK' COMMAND

The 'PEEK' function takes an address as its argument and delivers the contents of that address in the form of a number. For instance, if you reset the machine and type

PRINT PEEK(36879)

the machine replies with "27" because that is the equivalent of the bit pattern in memory cell 36879. As you will remember, cell 36879 is used to control the frame and background colours. You can alter its contents by a POKE command, and the PEEK will return the new value.

You can apply the PEEK function to any of the 65536 different addresses. If you select an address which isn't occupied (for instance, 1ØØØØ) the reply is meaningless. If you choose an area with ROM, you can find out what pattern was put in at the time of manufacture.

To get another view of the way bytes are used, let's have a look at the contents of cells 32776 to 32783. (This is part of the character ROM.) If you do a series of PEEKs, you get

```
32776 :  24
32777 :  36
32778 :  66
32779 : 126
32780 :  66
32781 :  66
32782 :  66
32783 :   Ø
```

This seems pretty meaningless, but look what happens when you convert back to the 'binary' form:

```
 24 : ØØØ11ØØØ        11
 36 : ØØ1ØØ1ØØ       1  1
 66 : Ø1ØØØØ1Ø      1    1
126 : Ø111111Ø      111111
 66 : Ø1ØØØØ1Ø      1    1
 66 : Ø1ØØØØ1Ø      1    1
  Ø : ØØØØØØØØ
```

The letter "A" pattern is now clear. When the VIC displays an "A" on the screen it uses the contents of these eight bytes to get the correct shape of the letter.

# EXPERIMENT 23·1



(a) Using PEEKs, find out what character is stored in location 33192 to 33199.

(b) Use the subroutine below in a program which explores the character ROM and shows how the 'shapes' are constructed. Remember that you can put a PEEK in a labelled command just like any other functions.

```
1ØØØ REM GIVEN A LOCATION IN X1, WORK
     OUT THE CORRESPONDING BINARY
     PATTERN AND DISPLAY IT
1Ø1Ø YY=256: FOR K=1 TO 8
1Ø15 YY=YY/2
1Ø2Ø IF X1>=YY THEN X1=X1−YY:
     PRINT"★";:GOTO 1Ø4Ø
1Ø3Ø PRINT " ";
1Ø4Ø NEXT K
1Ø5Ø PRINT: RETURN
```

| | |
|---|---|
| *Experiment 23.1 Completed* | |

## THE POKE COMMAND

The POKE command uses two arguments: an address (in the range Ø-65535) and a number (in the range Ø-255). The effect is to store the binary pattern of that number in the selected address. Of course this only works if the address is RAM or a controller, and it may have weird effects if you choose the address wrongly.

One important use for POKE commands is in driving the sound and picture controller. If you look at Figure 23.1, you'll see that the controller is connected to *four* of the modules in the address space:

(a) The vision and sound registers

(b) The Screen RAM

(c) The Colour RAM

(d) The Character ROM.

The picture you see on the screen is drawn afresh 50 times every second. Each time round, the generator looks at cell 36879, and paints the frame and background colours accordingly. Then it paints each of the 5Ø6 characters on the screen (we count 'space' as a character), starting at the top left and working from left to right and from the top down.

The process used to produce each of the characters is quite complex. To paint the first character, here's what happens:

First, the generator looks in address 768Ø, which is the *first* address of the screen RAM. Here it finds a screen *code* which tells it what character is needed. The code is shown in Figure 23.2, where you should ignore the column marked "SET2" for the present. Using this code sheet, you will see that an 'M' is represented by 13, or a ♥ by 83.

The table only goes up to 127, because the codes 128 to 255 stand for all the same characters in reverse video. For example, the code for a reversed \$-sign is 36+128 or 164.

Next, the generator goes to the character ROM to find out what shape to paint. To find the right shape, it multiplies the screen code by 8, adds on 32768, and fetches 8 bytes starting with the calculated address. For instance, it gets the shape of the 'M' from the 8 bytes starting at 32768+13★8 or 32872.

Next, the generator goes to the colour RAM and fetches the byte at 384ØØ. This tells it the *colour* of the first character according to the code

| Ø | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Black | White | Red | Cyan | Purple | Green | Blue | Yellow |

Finally, the generator now has enough information to paint the first character the right shape and colour.

When the first character is done, the generator displays the second character in the same way. This time, however, it uses cell 7681 (instead of 768Ø) to get a screen code and 384Ø1 (instead of 384ØØ) for the colour code.

The generator continues this way, until it has worked through all 5Ø6 bytes of the screen RAM and the colour RAM, and painted the whole picture. Then it starts all over again.

This system sounds complicated, but it is very flexible. The generator runs all the time and works quite independently of the microprocessor. To display any information, all that is necessary is to record the appropriate codes in the Screen and Colour RAMs. As soon as this is done the new character appears on the screen within 1/5Ø of a second.

Suppose you want to display a red diamond 7 lines down and 14 spaces across. Each whole line accounts for 22 characters, so this position is displayed 6★22+13 or 145 characters after the top left one. The corresponding cells are

768Ø+145 = 7825 in the Screen RAM

and 384ØØ+145 = 38545 in the Colour RAM.

According to Figure 23.2, the code for a diamond is 9Ø, and "Red" is 2, so the following pair of commands should put a red diamond in the right place:

```
POKE 7825,9Ø:POKE 38545,2
```

The diagrams in Figure 23.3 will help you work out the exact RAM address for any place on the screen. It is sometimes more convenient to draw pictures using POKEs than PRINT commands. For example, here is a program that draws a red diagonal line up the screen:

```
1Ø FOR J=21 TO 462 STEP 21
2Ø POKE 768Ø+J,78: POKE 384ØØ+J,2
3Ø NEXT J
4Ø GOTO 4Ø : REM LOOP STOP
```

## INTRODUCING ANIMATION

POKEs really become useful when you begin to think of animation. Let's suppose you want to put a moving circle on the screen. Basically the method consists of drawing the circle in one position, then wiping it out and drawing it in the next position, and so on. If you used PRINT statements with cursor movements this would be terribly clumsy; but with POKE's it is quite easy. Here, for example, is a short program which moves a circle about at random, but never letting it off the edge of the screen. Notice that X and Y represent the positions of the circle in places across and places down, respectively:

## SCREEN CODES

| SET 1 | SET 2 | POKE |
|---|---|---|
| @ | | Ø |
| A | a | 1 |
| B | b | 2 |
| C | c | 3 |
| D | d | 4 |
| E | e | 5 |
| F | f | 6 |
| G | g | 7 |
| H | h | 8 |
| I | i | 9 |
| J | j | 1Ø |
| K | k | 11 |
| L | l | 12 |
| M | m | 13 |
| N | n | 14 |
| O | o | 15 |
| P | p | 16 |
| Q | q | 17 |
| R | r | 18 |
| S | s | 19 |
| T | t | 2Ø |

| SET 1 | SET 2 | POKE |
|---|---|---|
| U | u | 21 |
| V | v | 22 |
| W | w | 23 |
| X | x | 24 |
| Y | y | 25 |
| Z | z | 26 |
| [ | | 27 |
| £ | | 28 |
| ] | | 29 |
| ↑ | | 3Ø |
| ← | | 31 |
| SPACE | | 32 |
| ! | | 33 |
| " | | 34 |
| # | | 35 |
| $ | | 36 |
| % | | 37 |
| & | | 38 |
| ' | | 39 |
| ( | | 4Ø |
| ) | | 41 |

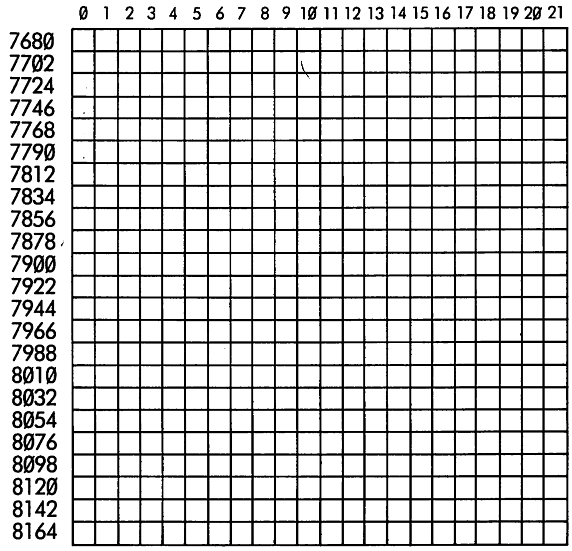
| SET 1 | SET 2 | POKE |
|---|---|---|
| ★ | | 42 |
| + | | 43 |
| , | | 44 |
| — | | 45 |
| . | | 46 |
| / | | 47 |
| Ø | | 48 |
| 1 | | 49 |
| 2 | | 5Ø |
| 3 | | 51 |
| 4 | | 52 |
| 5 | | 53 |
| 6 | | 54 |
| 7 | | 55 |
| 8 | | 56 |
| 9 | | 57 |
| : | | 58 |
| ; | | 59 |
| < | | 6Ø |
| = | | 61 |
| > | | 62 |

*Figure 23.2*

| SET 1 | SET 2 | POKE |
|---|---|---|
| ? | | 63 |
| | | 64 |
| ♠ | A | 65 |
| | B | 66 |
| | C | 67 |
| | D | 68 |
| | E | 69 |
| | F | 70 |
| | G | 71 |
| | H | 72 |
| | I | 73 |
| | J | 74 |
| | K | 75 |
| | L | 76 |
| | M | 77 |
| | N | 78 |
| | O | 79 |
| | P | 80 |
| ● | Q | 81 |
| | R | 82 |
| ♥ | S | 83 |
| | T | 84 |
| | U | 85 |
| | V | 86 |
| ○ | W | 87 |
| ♣ | X | 88 |
| | Y | 89 |
| ♦ | Z | 90 |
| | | 91 |
| | | 92 |
| | | 93 |
| π | | 94 |
| | | 95 |
| SPACE | | 96 |
| | | 97 |
| | | 98 |
| | | 99 |
| | | 100 |
| | | 101 |
| | | 102 |
| | | 103 |
| | | 104 |
| | | 105 |
| | | 106 |
| | | 107 |
| | | 108 |
| | | 109 |
| | | 110 |
| | | 111 |
| | | 112 |
| | | 113 |
| | | 114 |
| | | 115 |
| | | 116 |
| | | 117 |
| | | 118 |
| | | 119 |
| | | 120 |
| | | 121 |
| | √ | 122 |
| | | 123 |
| | | 124 |
| | | 125 |
| | | 126 |
| | | 127 |

```
1Ø PRINT " [SHIFT] and [CLR HOME] "
2Ø X=12 : Y = 1Ø
3Ø XN=X+INT(3★RND (Ø))−1
4Ø IF XN<Ø OR XN > 21 THEN 3Ø
5Ø YN=Y+INT(3★RND(Ø))−1
6Ø IF YN<Ø OR YN>22 THEN 5Ø
7Ø POKE 768Ø+22★Y+X, 32
8Ø Y=YN: X=XN
9Ø POKE 768Ø+22★Y+X, 87
1ØØ POKE 384ØØ+22★Y+X, Ø
11Ø GOTO 3Ø
```

**SCREEN RAM MAP**




**COLOUR RAM MAP**

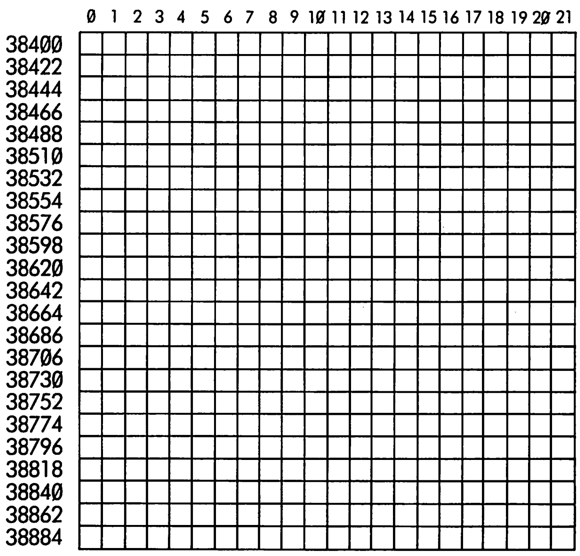



*Figure 23.3*

# EXPERIMENT 23·2



Here is a slightly longer program which uses POKE's to create an "artistic" effect. Try it out, and then do your best to improve on it.

```
1Ø PRINT " [SHIFT] and [CLR HOME] "
2Ø FOR K=Ø TO 7
3Ø FOR J=K TO 9
4Ø Y1=2+J
5Ø FOR X1=2+J TO 2Ø−J
6Ø GOSUB 1ØØØ
7Ø NEXT X1
8Ø X1=2Ø−J
9Ø FOR Y1=2+J TO 2Ø−J
1ØØ GOSUB 1ØØØ
11Ø NEXT Y1
12Ø Y1=2Ø−J
13Ø FOR X1=2Ø−J TO 2+J STEP−1
14Ø GOSUB 1ØØØ
15Ø NEXT X1
16Ø X1=2+J
17Ø FOR Y1=2Ø−J TO 2+J STEP−1
18Ø GOSUB 1ØØØ
19Ø NEXT Y1
2ØØ NEXT J
21Ø NEXT K
22Ø GOTO 1Ø
1ØØØ REM SUBROUTINE TO DRAW ★ AT
     X1, Y1 COLOUR K
1Ø1Ø ZZ=X1+22★Y1
1Ø2Ø POKE 768Ø+ZZ, K
1Ø3Ø POKE 384ØØ+ZZ, K
1Ø4Ø RETURN
```

| *Experiment 23.2 Completed* | |
|---|---|

## MORE ABOUT PEEKS AND POKES

A common way of using PEEK is to examine what is on the screen. If you give a PEEK with the address of a cell in the screen RAM, the result will be the screen code of the character in that cell. This facility is useful if you want to draw a picture on the screen using the cursor and colour controls, and then to analyse or record the picture with a program.

To let yourself draw the picture without interference, write commands to clear the screen and input X (or any other variable). When the ? comes up, you can use the cursor commands to draw any picture you like; you can even rub out the input command ?. On the other hand you must not use RETURN until the picture is finished; and when you do press RETURN the cursor must be somewhere in a completely blank line, preferably above or below your picture.

All this is illustrated in the following program, which counts and displays the number of non-space characters on the screen. Key it in, start it, and then use the cursor control to scatter a few symbols all over the screen. Then strike RETURN.

```
1Ø INPUT " SHIFT and CLR HOME ";X : REM X
   IS A DUMMY VARIABLE
2Ø S=Ø
3Ø FOR J=Ø TO 5Ø5 : REM SCAN SCREEN
4Ø IF PEEK(768Ø+J)<>32 THEN S=S+1 :
   REM 32 IS SCREEN CODE FOR SPACE
5Ø NEXT J
6Ø PRINT "NUMBER OF SYMBOLS ="; S
7Ø STOP
```

There are certain other locations which can be conveniently POKE'd to change the behaviour of the VIC in predictable and useful ways.

Address 36869 controls the selection of character shapes. The command

```
POKE 36869, 242
```

will switch you to SET 2 in Figure 23.2. Here you have both upper and lower case letters, but far fewer graphics. The set is useful for presenting readable reports and in other types of business Data Processing. It is impossible to show SET 1 and SET 2 characters on the screen at the same time; this is why so many characters are duplicated.

If you want to write a program which displays its results in SET 2, then you should plan this well in advance. Before you start keying the program in, press the SHIFT and C= keys together. Anything you type will now appear in SET 2; letters will be displayed in lower case unless you hold the shift key down. The commands of your BASIC program and your REM's must all be in lower case, but you can put upper case letters in strings if necessary. Here is an example:

(press SHIFT and C= )

```
1Ø poke 36869,242 : rem set output to set 2
2Ø print "This is a Set 2 Display"
3Ø stop
```

To switch back to SET 1 (which you can do any time) give the command

```
poke 36869,24Ø
```

or press SHIFT and C= again.

There is a group of locations which control the behaviour of the keyboard.

★ Repeating keys: When the VIC is switched on, the only keys which 'repeat' if you hold them down are space and the cursor control keys. This is controlled by the contents of address 65Ø, which is interpreted as follows:

```
  Ø : Only the 'standard' keys repeat
127 : No keys repeat
128 : All keys repeat.
```

If you give the command

```
POKE 65Ø, 128
```

you'll find that all the keys are now 'repeaters'.

★ Keyboard queue: If you type characters faster than your program can accept them, they will be put in a 'queue' and delivered to your program one at a time. The number of characters in the queue can be inspected by PEEKing 198. The characters in the queue can also be thrown away by POKEing Ø into address 198. This is often useful in games where the aim is to make the computer react to what the player is doing now, not what he may mistakenly have done a few seconds ago.

## AN EXAMPLE OF ANIMATION

We end this unit by discussing the design of an animated game. Load the program entitled 'WASPS', and play it several times, until you have gained some skill as a slayer of wasps.

The whole WASP program is reproduced at the end of this unit, and we'll have a look at its general design principles.

The WASP game, like most other computer games, is a simulation or if you like an imitation, of something which is supposed to be happening in the 'outside world'. In this game the program simulates a hunter in a room full of wasps. The wasps move at random, whereas the hunter moves, turns and fires his fly-spray in response to the keyboard. Various events can happen:

(a) A wasp may be killed

(b) The hunter may be stung

(c) The hunter may run out of fly-spray.

The framework of the program is a *model*, or set of variables, which completely describes the position at any instant. Once this model has been designed, you can write various pieces of code (mostly subroutines) which operate on the model and change it in accordance with the events supposed to happen in the outside world.

Here then, is a description of the model in the WASP game:

N : Number of wasps at the start

NA : Number of wasps left at any moment

TT : Starting time of hunt (in jiffies)

BU : Number of shots of fly-spray left

ST : Number of times hunter has been stung

IP : Wasp drone pitch (the pitch of the sound made by the wasps)

A,B : The present position of the hunter. A is the number of screen columns from the left, and B the number of rows from the top.

C : The present direction of the hunter:
1=North,
2=North-East,
3=East,
4=South-East,
5=South,
6=South-West,
7=West,
8=North-West.

The position of each *wasp* is recorded as two elements in the array W%(N,2). Thus the column position of the first wasp is in W%(1,1), and its row position is W%(1,2). The second wasp occupies W%(2,1) and W%(2,2), and so on.

The position of the last *active* wasp is stored in W%(NA,1) and W%(NA,2). If a wasp (other than the last active one) is killed, the record of all the ones below it are moved up to fill its place.

For example:

NA=6

W%

| | | |
|---|---|---|
| 1 | 17 | |
| 3 | 12 | |
| 18 | 2 | |
| 10 | 7 | ←This one killed |
| 7 | 9 | |
| 4 | 17 | |

gives

NA=5

W%

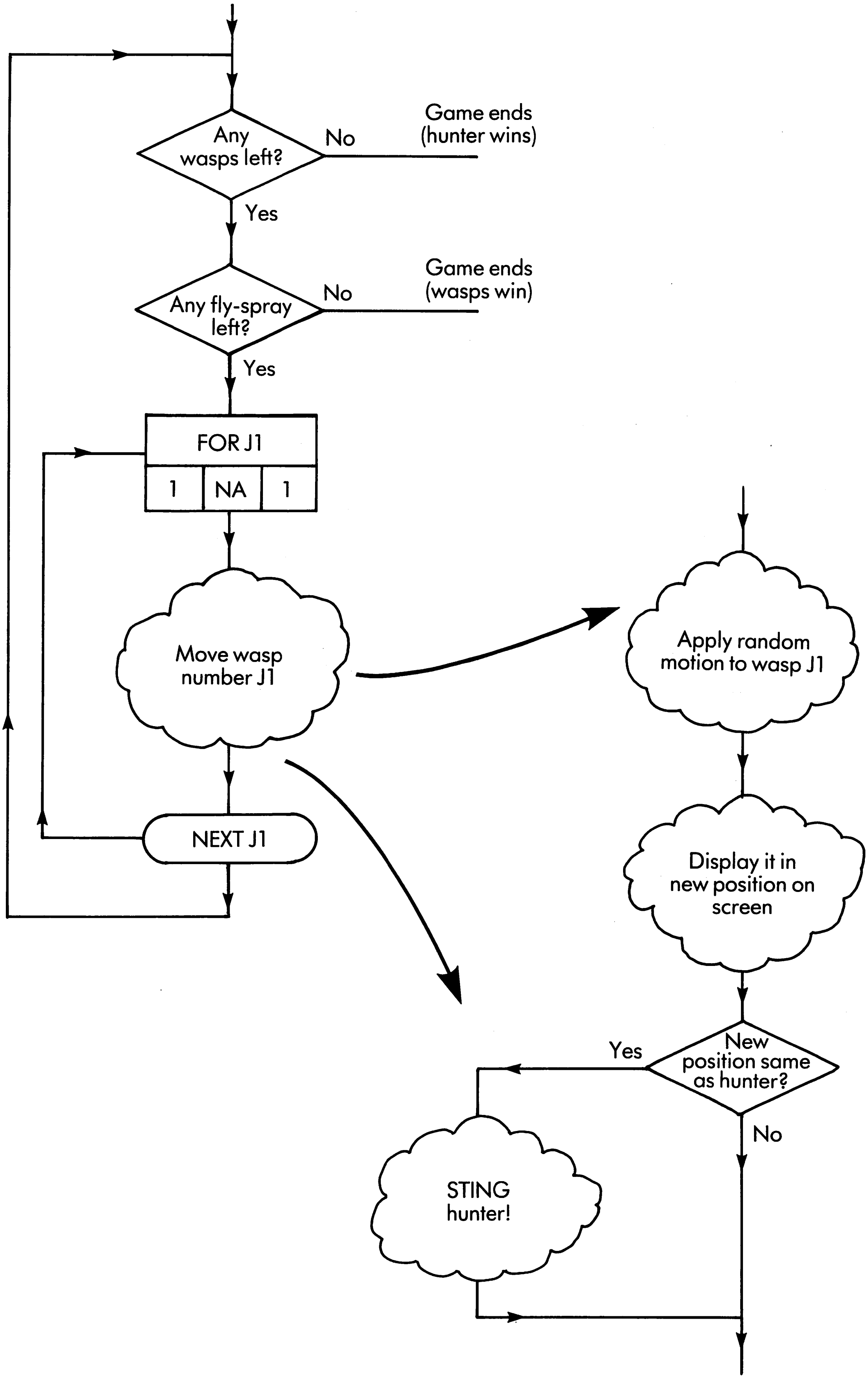
| | | |
|---|---|---|
| 1 | 17 | |
| 3 | 12 | |
| 18 | 2 | |
| 7 | 9 | |
| 4 | 17 | |
| — | — | This row unused. |



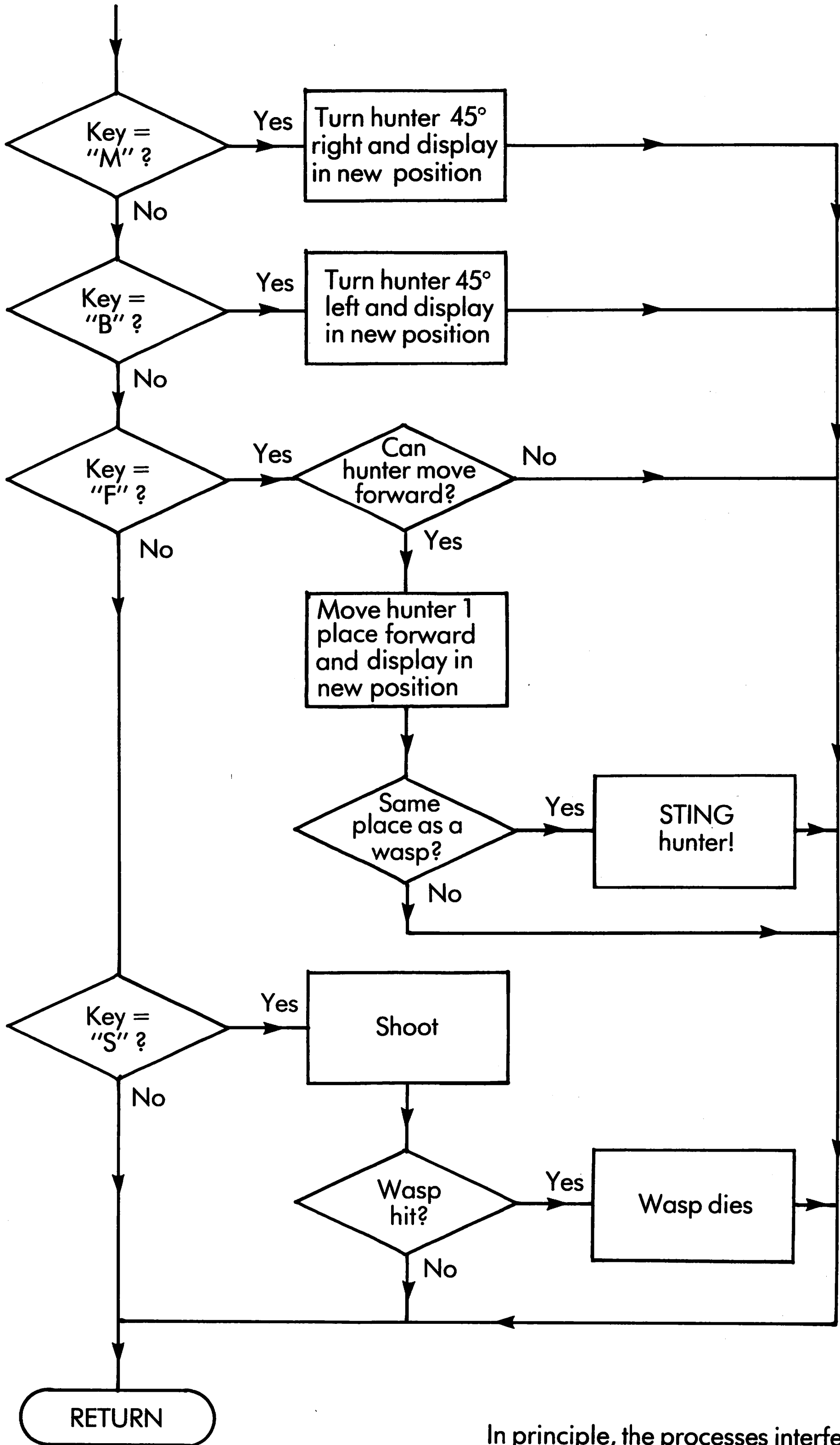
The game itself is organised as two 'processes', which run almost independently: the 'wasp' process and the 'hunter' process.

The 'wasp' process is responsible for making all the wasps move about. One by one, each wasp in the list is shifted at random by at most one cell, with the restriction that it mustn't fall off the edge of the screen. If a wasp moves to the same place as the hunter, it stings him.

The wasp process runs over and over again until *either* there are no wasps left, *or* the fly-spray runs out. Its flow chart is:




The 'hunter' process is called whenever a key is pressed. Its flow chart is:




In principle, the processes interfere with each other only in quite specific ways. For example, the activities of the hunter gradually reduce the number of active wasps *and* the amount of ammunition remaining, eventually forcing the process to stop.

In practice, we have to arrange for both processes to run at the same time (more or less). A simple way to do this is to give the hunter a chance to do something after each wasp has moved. A character is fetched from the keyboard, and if a key has been pressed the 'hunter process' is allowed to cycle.

We can now give a detailed description of the program. It fits comfortably into the 3 1/2K VIC store with about 5ØØ bytes left over for modifications and improvements. To win this space some space-saving measures have been used.

Lines 1Ø-9Ø display a set of user instructions

Line 1ØØ sets up certain numbers in symbolic form. This will help to save space later; for example each reference to "PE" will be 3 bytes shorter than a reference to "36879".

Lines 12Ø-13Ø determine the number of wasps to start with.



Line 14Ø declares four arrays. They are all integer arrays to save space. W% is, as you know, used to hold the position of each active wasp. V%, U% and D% all help with displaying and moving the hunter.

You will remember that the squares in the screen RAM are numbered from left to right, starting with the top row and working down.

| | | | |
|---|---|---|---|
| | 7930 | | |
| 7951 | 7952 | 7953 | 7954 |
| 7973 | 7974 | 7975 | 7976 |
| 7995 | 7996 | 7997 | |
| | 8018 | | |

Here is a part of the screen, where each cell is marked with its address. Suppose the hunter is at some cell with his aim pointing due North. As you can see, the fly-spray can will occupy a square numbered 22 *less* than the one he occupies himself. Likewise, if he points North-East, the 'aim' cell will be numbered 21 less, and so on. Array V% holds the "cell number difference" for each of the 8 possible directions, starting with V%(1) (North) and working round to V%(8) (North-East). In fact (bearing in mind that variable C holds the direction the hunter is facing) the appropriate cell number difference is always V%(C).

Array V% holds the screen codes of the symbols used to represent the hunter's aim. They vary according to direction:

| N | NE | E | SE | S | SW | W | NW |
|---|---|---|---|---|---|---|---|
| \| | / | — | \ | \| | / | — | \ |

Using these tables, the hunter can easily be displayed in any position and any direction. Look at lines 2ØØØ-2Ø2Ø, remembering that the hunter's position is recorded in variables A and B. SR is the constant 768Ø, which is the address of the first cell in the screen RAM.

Array D% holds the movements East and South which correspond to a move in any of the eight directions. The values are:

| | | |
|---|---|---|
| N | Ø | −1 |
| NE | 1 | −1 |
| E | 1 | Ø |
| SE | 1 | 1 |
| S | Ø | 1 |
| SW | −1 | 1 |
| W | −1 | Ø |
| NW | −1 | −1 |

This makes it easy to move the hunter. For instance, to go one square in direction 6 (SW) we add D%(6,1) to A and D%(6,2) to B.

Lines 15Ø to 21Ø set up the appropriate values for V%, U% and D%.

Line 22Ø sets up the initial supply of fly-spray. The formula ensures a sliding scale like this

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| No. of Wasps | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| No. of shots | 7 | 9 | 12 | 14 | 15 | 17 | 18 | 19 |

This allows for the fact the first few wasps out of a large number are much easier to catch than the last few.

Line 24Ø sets the whole screen colour to 'black'.

Lines 25Ø to 29Ø work out initial positions for all the wasps, and display them on the screen. At first, all the wasps are put in the top half of the screen (lines 1 to 12).

Line 32Ø fixes the starting position of the hunter and displays him in this position.

Lines 33Ø to 42Ø form the heart of the simulation; they run the wasp process and call the hunter process whenever a key is pressed. 34Ø and 35Ø display the current state of affairs. Note that the subroutine at 1ØØØ moves wasp J1, and the one at 3ØØØ actuates the hunter.

Lines 5ØØ to 57Ø display the final messages of congratulations or consolation, as may be appropriate.

The wasp-moving subroutine is at lines 1000 to 1090. The basic method is to get the present position of the wasp into the local variables XN and YN. Each of these numbers is then 'perturbed' by a random amount which may be +1, 0 or −1. If the result is outside the range of the screen it is rejected and the process tried again.

When a new position has been determined, the wasp at the old position is erased (line 1040). A wasp is painted at the new place unless that position is already occupied by something else. Then the new position is recorded in the table at W%(J1,1) and W%(J1,2).

At this point the wasp 'drone' is generated. The current 'pitch' of the drone is stored in variable IP. The value of IP is perturbed by one unit at random, with a trap to prevent it from straying too far from its normal range of about 192. Then the variable is used to start a droning note which continues until the next time IP is changed.

Finally, the new position of the wasp is compared with that of the hunter, and if they are the same the 'sting' subroutine at 4000 is called.

Lines 2000 to 2020 display the hunter at his new position,

and

Lines 2500 to 2520 delete the hunter from an old position by painting spaces at the appropriate positions.

Lines 3000 to 3270 look after the hunter. The parts of this subroutine are as follows:
3020-3030 Turn right. Note that North (C=1) must follow North-East (C=8).
3050-3060 Turn left. Note that North-East (C=8) must follow North (C=1).
3080-3140 Move forward. A *tentative* new position is produced in AA and BB, and is only transferred to A and B if it is not too near the edge of the screen.
When the move is made, the list of wasp positions is searched to see if the hunter has sat on a wasp; if so, he is stung. This happens in lines 3110 to 3130.
3160-3270 Shoot. PP and QQ are the positions of the target area. Lines 3170 and 3190 to 3240 are concerned with the effects (sound and vision), of the shot. 3250 to 3270 search the wasp list to see if a hit has occurred. If so, the subroutine at 5000 is called.

Lines 4000 to 4080 is a subroutine called when the hunter is stung. It is mainly audio-visual effects, but there are also arrangements for the hunter to jump to a new random position.

Lines 5000 to 5070 handle the death of the wasp. Apart from the usual effects, the record of the dead wasp is removed from the list and the other records moved up if necessary.

# EXPERIMENT 23·3



Design and program your own animated game. Possible areas of interest include:

Shooting aliens from outer space.

Searching a maze with a monster chasing you

Catching randomly-thrown balls.

| *Experiment 23.3 Completed* | |
|---|---|

```
10 REM WASPSHOOTER COPYRIGHT
     (C) ANDREW COLIN 1981
20 PRINT" [SHIFT] and [CLR HOME] [CRSR ↑↓]
     W A S P S H O O T E R":PRINT:PRINT
30 PRINT"KILL ALL THE WASPS"
40 PRINT"BEFORE THE FLY-SPRAY"
50 PRINT "RUNS OUT"
60 PRINT: PRINT" M TO TURN RIGHT"
70 PRINT" B TO TURN LEFT"
80 PRINT" F TO GO FORWARD"
90 PRINT" S TO SHOOT":PRINT
100 PA=36875:PB=36876:PC=36877:
     PD=36878:PE=36879:SR=768Ø
110 POKE PB,Ø:POKE PC,Ø:POKE PD,15
120 INPUT"HOW MANY WASPS";N
130 IF N<1 OR N>2Ø THEN PRINT "1 TO
     2Ø PLEASE":GOTO 12Ø
140 DIM V%(8),U%(8),D%(8,2),W%(N,2)
150 FOR J = 1 TO 8
160 READ V%(J),U%(J),D%(J,1),D%(J,2)
170 NEXT J
180 DATA -22,93,Ø,-1,-21,78,1,-1
190 DATA 1,67,1,Ø,23,77,1,1
200 DATA 22,93,Ø,1,21,78,-1,1
210 DATA -1,67,-1,Ø,-23,77,-1,-1
220 BU=INT(7★SQR(N)): SQ=Ø
230 PRINT " [SHIFT] and [CLR HOME] "
240 FOR J=Ø TO 5Ø5: POKE 384ØØ+J,Ø:
     NEXT J
250 FOR J = 1 TO N
260 W%(J,1)=INT(22★RND(Ø))
270 W%(J,2)=INT(12★RND(Ø))+1
280 POKE SR+22★W%(J,2)+W%(J,1),35
290 NEXT J
300 NA = N
310 TS = TI
320 A=3:B=18:C=2: GOSUB 2ØØØ
330 IF NA=Ø THEN 5ØØ
335 IF BU = Ø THEN 6ØØ
340 PRINT" [CLR HOME] "
350 PRINT" [CLR HOME] WASPS";NA;"TIME";
     INT((TI-TS)/6Ø)
360 PRINT "SHOTS 5 spaces [SHIFT] and
     [CRSR ⇐⇒] [SHIFT] and [CRSR ⇐⇒] [SHIFT] and
     [CRSR ⇐⇒] [SHIFT] and [CRSR ⇐⇒] [SHIFT]
     and [CRSR ⇐⇒] "; BU; " 4 spaces "
370 FOR J1 = 1 TO NA
380 IF W%(J1,1)>=Ø THEN GOSUB 1ØØØ
390 GET A$: IF A$ = " " THEN 41Ø
400 GOSUB 3ØØØ
410 NEXT J1
420 GOSUB 2ØØØ:GOTO 33Ø

500 REM WINS
510 PRINT" [SHIFT] and [CLR HOME] [CRSR ↑↓]
     [CRSR ↑↓] 3 spaces  WELL DONE !":PRINT
520 PRINT" YOU HAVE KILLED "; N-NA:
     PRINT
530 PRINT"WASPS IN";INT((TI=TS)/6Ø);
     "SECONDS":PRINT
540 PRINT"YOU WERE STUNG":PRINT
550 PRINT SQ;"TIMES"
560 POKE PD,Ø: POKE PC,Ø:POK[illegible] Ø:
     POKE PA,Ø
570 STOP

600 REM OUT OF FLY-SPRAY
610 PRINT" [SHIFT] and [CLR HOME]
     [CRSR ↑↓] [CRSR ↑↓] SORRY—NO FLY-
     SPRAY":PRINT
620 PRINT"LEFT!":PRINT
630 GOTO 52Ø

1000 REM MOVE J1'TH WASP AT RANDOM
1010 XX=W%(J1,1): YY=W%(J1,2)
1020 XN=XX+INT(3★RND(Ø))-1: IF XN<Ø
      OR XN>21 THEN 1Ø2Ø
1030 YN=YY+INT(3★RND(Ø))-1: IF YN<1
      OR YN>22 THEN 1Ø3Ø
1040 POKE SR+22★ YY+XX,32
1050 ZZ=SR+22★YN+XN:IF PEEK (ZZ)=32
      THEN POKE ZZ,35
1060 W%(J1,1)=XN:W%(J1,2)=YN
1070 IP=IP+INT(3★RND(Ø))-1: IF IP<18Ø
      OR IP>2Ø5 THEN IP=192
1080 IF XN = A AND YN = B THE[illegible]B
      4ØØØ
1090 POKE PA, IP: RETURN

2000 REM DISPLAY HUNTER
2010 XX=SR +22★B+A: YY=XX[illegible]
2020 POKE XX,81: POKE YY,U%(C).
      RETURN

2500 REM ERASE HUNTER
2510 XX=SR+22★B+A:YY=XX+V%(C)
2520 POKE XX,32: POKE YY,32: RETURN

3000 REM MOVE HUNTER OR SHOOT
3010 IF A$<>"M" THEN 3Ø4Ø
3020 GOSUB 25ØØ: C=C+1: IF C[illegible]
      C=1
3030 GOSUB 2ØØØ: RETURN
3040 IF A$ <>"B" THEN 3Ø7Ø
3050 GOSUB 25ØØ:C=C-1: IF C=Ø THEN
      C=8
3060 GOSUB 2ØØØ: RETURN
3070 IF A$<>"F" THEN 315Ø
3080 GOSUB 25ØØ: AA=A+D%(C,1):
      BB=B+D%(C,2)
3090 IF AA>2 AND AA <19 AND BB>2
      AND BB<19 THEN A=AA:B=BB
3100 GOSUB 2ØØØ
3110 FOR JJ= 1 TO NA
3120 IF A=W%(JJ,1) AND B=W%(JJ,2)
      THEN GOSUB 4ØØØ
3130 NEXT JJ
3140 RETURN
3150 IF A$<>"S" THEN RETURN
3160 PP=A+2★D%(C,1):QQ=B+2★D% (C,2)
```

```
3170 POKE PC,252
3180 BU=BU-1
3190 RR=SR + 22*QQ+PP
3200 FOR KK = 1 TO 5
3210 POKE RR,102: FOR TT= 1 TO 30:
     NEXT TT
3220 POKE RR,32: FOR TT= 1 TO 50:
     NEXT TT
3230 NEXT KK
3240 POKE PC,0
3250 FOR JJ = 1 TO NA
3260 IF PP=W%(JJ,1) AND QQ=W%(JJ,2)
     THEN J2=JJ: GOSUB 5000
3270 NEXT JJ : RETURN

4000 REM HUNTER IS STUNG
4010 PRINT" [CLR HOME] S T U N G ! ! !"
4020 GOSUB 2500
4030 A=INT(3+16*RND(0)):
     B=INT(3+16*RND(0)):
     C=INT(1+8*RND(0))
4040 POKE PD,15: GOSUB 2000:
     SQ=SQ + 1
4050 FOR JJ=1 TO 20: POKE PB,240-JJ:
     POKE PE,250-JJ
4060 FOR TT = 1 TO 50: NEXT TT
4070 NEXT JJ
4080 POKE PB,0: POKE PE,27: RETURN

5000 REM WASP IS KILLED
5010 PRINT" [CLR HOME] A WASP BITES THE
     DUST!"
5020 POKE PD,15: POKE PE,123
5030 FOR JJ = 1 TO 20: POKE PB,255-JJ
5040 FOR TT= 1 TO 10: NEXT TT
5050 POKE PB,0: FOR TT = 1 TO 40: NEXT TT
5060 NEXT JJ: POKE PE,27
5070 IF J2 = NA THEN NA=NA-1: RETURN
5080 W%(J2,1) = W%(J2+1,1)
5090 W%(J2,2) = W%(J2+1,2)
5100 J2=J2+1: GOTO 5070
```

# UNIT:24

247

## MORE ABOUT LOGICAL OPERATORS

In this Unit we complete our study of VIC BASIC by considering a few miscellaneous topics.

Unit 17 looked at the use of the logical operators AND, OR and NOT in building compound conditions. These same operators can also be used in a completely different context to manipulate the binary digits in numbers and other variables.

Type the command

PRINT 13 AND 17

The result, 5, is completely mysterious until you look at the *binary* representation of the numbers involved:

```
13 = ....0001101
 7 = ....0000111
----------------
 5 = ....0000101
```

As you can see, the result has a '1' only in the columns where *both* the original numbers have '1's. We can explain the AND operation with a 'truth table' which applies independently to each column in the sum:

```
0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1
```

Using this table, we can predict the result of the command

PRINT 27 AND 6

```
27 = ...0011011
 6 = ...0000110
----------------
     ...0000010 = 2
```

To make sure that you have understood the AND operation, try working out in advance the results of the following:

```
PRINT 15 AND 12
PRINT 21 AND 10
PRINT 11 AND 7
```

Check your calculation on the VIC in each case.

The OR operator is very like AND; the difference is that it gives a '1' in any column where *either* or *both* the original numbers have '1's. Its truth table is

```
0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1
```

Using this table, you should have no trouble predicting the result of the command

PRINT 7 OR 10

The NOT operator just takes a single number and changes every bit to its opposite. You will find that

NOT (....0001010) = ....1110101

In the VIC (and indeed in most other computers) a number which consists *entirely* of '1's represents −1 (negative 1). As you would expect, the result of the command

PRINT NOT 0

is −1.

The AND, OR and NOT operations are useful in working with quantities where the individual binary digits have special meanings. For instance, if you are using the VIC to control the lights in your house through the user port, it is quite likely that the positions of eight separate switches will be represented as eight binary digits in a single number. To discover if, say the fifth switch from the right is on, you would do an AND operation between the character and the binary number ..0010000. The result would be different from zero only if the number had a '1' in that position — in other words, the 5'th switch was on.

The equivalent of ..0010000 is 16, so your control program might have a line which read

```
360 IF (S AND 16) <> 0 THEN 590
```

## HOW VIC EVALUATES CONDITIONS

You may be wondering how these apparent new meanings of the operators tie in with the conventional ones used for compound conditions. To discover the answer we must look a little deeper in the mechanisms of the VIC.

When the machine works out a simple condition (such as X=5 or A$<>"YES" or 5=4) it always produces a *truth value* which is −1 if the condition is true and 0 if it is false. Try the following commands (even though they look odd) and explain to yourself why they produce the results they do:

```
PRINT 5=4
PRINT 6<9
PRINT 9>6
PRINT (1=1)★(1<2)
```

The IF command always has an expression between IF and THEN. This is usually a condition, but it needn't be; forms like

IF X−3 THEN ...

are quite acceptable. The command (or group of commands) following THEN is executed if the expression after IF has any value *except* zero. Run the following program and explain its results:

```
10 FOR X = 1 TO 5
20 IF X-3 THEN PRINT X
30 NEXT X
40 STOP
```

Even when the expression *is* a condition, the way the machine works is still the same. Consider the command

```
IF "DONALD" < "MICKY" THEN PRINT
"PLUTO"
```

We see that the condition is true, and we expect that the machine will indeed display the string "PLUTO". The VIC actually goes through an intermediate stage: it first evaluates the condition to '1', and then it obeys the PRINT command because −1 is not the same as zero.

To complete the explanation of compound conditions, all that need be said is that the logical operators can be applied to the truth values produced by simple conditions. Suppose that X$="D". Then the compound condition inside the expression

```
IF NOT(X$ = "C" OR X$ = "D")
```

works out as

```
  NOT (0 OR −1)
= NOT (....00000 OR ....11111)
= NOT (....11111)
= ....00000
```

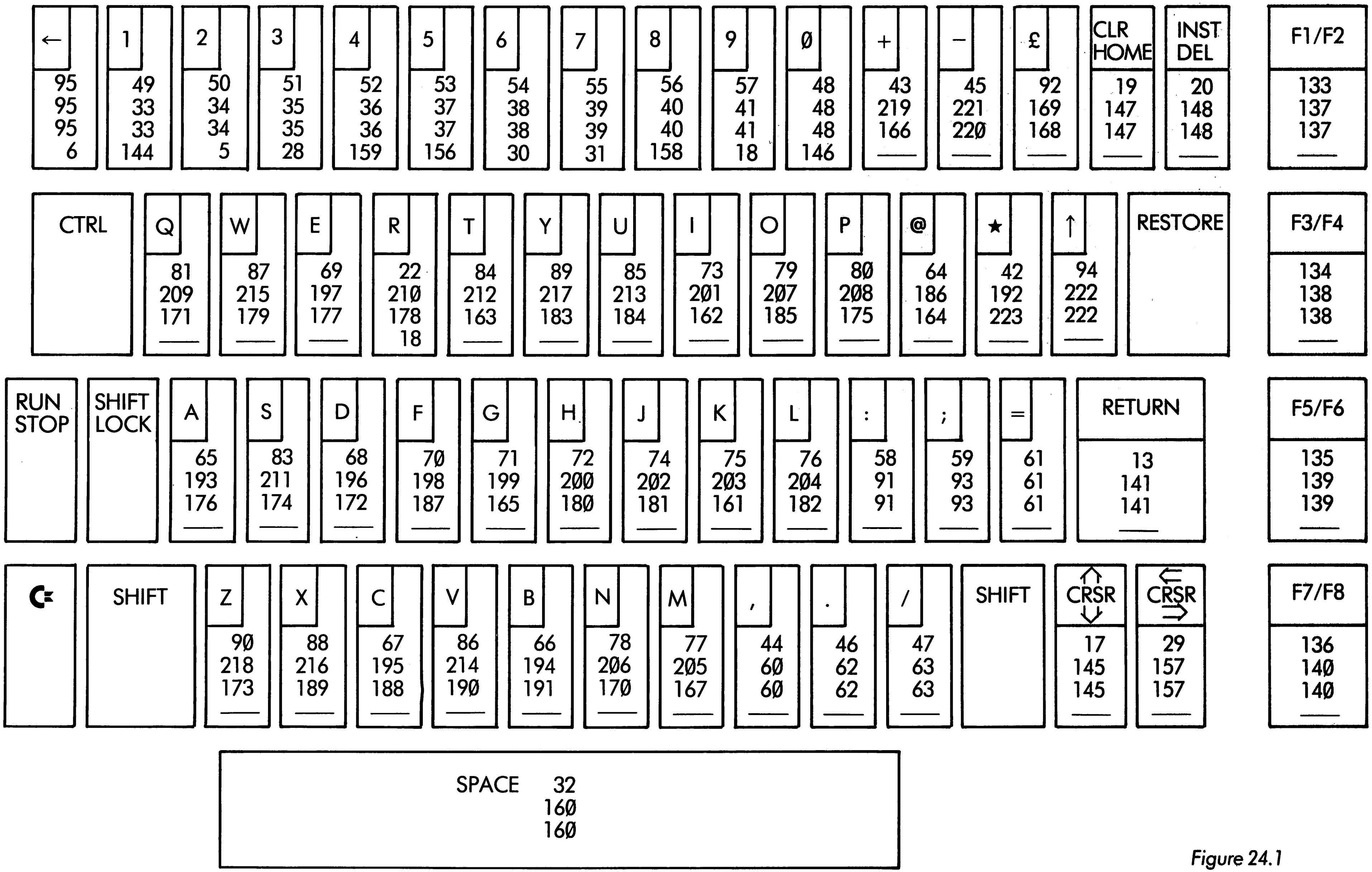
so the condition is *false*.

## CBM ASCII CODES

The next subject is the internal representation of characters. You already know that a string, when stored internally, takes up one byte for each character it contains. It is sometimes useful to know exactly how each character is represented.

In Unit 23 we introduced the idea of a 'screen code' and gave a table which showed how each character which could be displayed on the screen had its own special code. Inside the VIC, characters are also represented by a code, but it is a *different* one from the screen code! You can see that it must be different, because the code must be able to handle every character produced by pressing a key on the keyboard. This includes keys like RETURN or cursor movements which don't correspond to any displayable symbols.

The code used is a modification of the American Standard Code for Information Interchange, or "ASCII" for short. This code allows information to be moved over telephone lines between machines of various sorts.



← 95 95 95 6
1 49 33 33 144
2 50 34 34 5
3 51 35 35 28
4 52 36 36 159
5 53 37 37 156
6 54 38 38 30
7 55 39 39 31
8 56 40 40 158
9 57 41 41 18
Ø 48 48 48 146
+ 43 219 166
− 45 221 22Ø
£ 92 169 168
CLR HOME 19 147 147
INST DEL 20 148 148
F1/F2 133 137 137
CTRL
Q 81 209 171
W 87 215 179
E 69 197 177
R 22 21Ø 178 18
T 84 212 163
Y 89 217 183
U 85 213 184
I 73 2Ø1 162
O 79 2Ø7 185
P 8Ø 2Ø8 175
@ 64 186 164
★ 42 192 223
↑ 94 222 222
RESTORE
F3/F4 134 138 138
RUN STOP
SHIFT LOCK
A 65 193 176
S 83 211 174
D 68 196 172
F 7Ø 198 187
G 71 199 165
H 72 2ØØ 18Ø
J 74 2Ø2 181
K 75 2Ø3 161
L 76 2Ø4 182
: 58 91 91
; 59 93 93
= 61 61 61
RETURN 13 141 141
F5/F6 135 139 139
SHIFT
Z 9Ø 218 173
X 88 216 189
C 67 195 188
V 86 214 19Ø
B 66 194 191
N 78 2Ø6 17Ø
M 77 2Ø5 167
, 44 6Ø 6Ø
. 46 62 62
/ 47 63 63
SHIFT
CRSR ⇕ 17 145 145
CRSR ⇔ 29 157 157
F7/F8 136 14Ø 14Ø
SPACE 32 16Ø 16Ø


Figure 24.1

Details of the CBM ASCII code are given in Figure 24.1. This is a 'stretched' drawing of the keyboard, and shows the codes generated when the keys are struck. Each key has four (or three) numbers. They correspond to

a) The "unshifted" character

b) The "normal shift" character (key pressed when SHIFT held down)

c) The "Commodore Shift" character (key pressed when C= held down)

d) The "Control Shift" character (key pressed when CTRL held down).

Only some keys respond when CTRL is held down. Those that don't are marked with a —— below the other three numbers.

*(NOTE: Some of the values in this diagram are different from those shown in the table at the back of the manual supplied with your VIC. Those shown here are the ones generated by the ASCII command. Either set of values can be used with the CHR$ command.)*

The diagram shows you that — for example — when the D key is struck with C= held down, the CBM ASCII code of the character produced is 172. The string "COMMODORE" would be stored as a sequence of 9 bytes with the values 67, 79, 77, 77, 79, 68, 79, 82, 69. The diagram makes it plain that the cursor control keys and the special function keys at the right of the keyboard produce CBM ASCII codes, even though they don't correspond to any printed character.

The standard function which delivers the CBM ASCII code of any character is ASC. It takes a string as its argument and produces the CBM ASCII code of the *first* character. Thus

```
PRINT ASC("X")
```

will give 88 and

```
PRINT ASC("123456")
```

gives 49.

For obvious reasons, ASC can't be applied to the null string (" "). If you try, it gives an

```
?ILLEGAL QUANTITY ERROR.
```

The program used to fill in the numbers in Figure 24.1 was basically this:

```
1Ø GET A$:IF A$ = " " THEN 1Ø
2Ø PRINT A$; ASC(A$)
3Ø GOTO 1Ø
```

Key in this program and run it to check a few of the values in Figure 24.1. You'll need some ingenuity to handle the control characters; you could, for example, remove the A$; from line 2Ø.

## USING ASC — COUNTING LETTER OCCURENCES

The ASC function is particularly useful in two areas. The first is when you would like to translate individual characters into numbers. For instance, you may want to 'crack' a secret code by analysing a cryptic message and counting the number of times each letter is used. Clearly you could write a program which had lots of instructions like

```
...
IF A$ = "J" THEN AJ=AJ+1
IF A$ = "K" THEN AK=AK+1
...
```

and later,

```
...
PRINT "J",AJ
PRINT "K",AK
...
```

With the ASC function you can do much better than this. The diagram shows you that the CBM ASCII codes for the letters start at 65 for A and go up to 9Ø for Z. We can use the CBM ASCII code of each letter as a subscript for an array, where each element corresponds to one letter and keeps track of the number of times that letter is used.

In the program below, ★ is used as a terminating character. Other characters which are not letters are displayed on the screen but otherwise ignored. The program lets you type a message and then displays the frequency of each letter:

```
 1Ø DIM T(26)
 2Ø GET X$:IF X$=""THEN 2Ø
 3Ø IF X$="★" THEN 9Ø
 4Ø PRINT X$;
 5Ø IF X$ < "A" OR X$ > "Z" THEN 2Ø
 6Ø P = ASC(X$)-64
 7Ø T(P) = T(P)+1
 8Ø GOTO 2Ø
 9Ø PRINT
1ØØ FOR P = 1 TO 26
11Ø PRINT T(P);
12Ø NEXT P
13Ø STOP
```

**Glossary**

T(26): Array of counters. T(1) for A's, T(2) for B's, and so on up to T(26) for Z's.
X$: Current character
P: ASCII code of current character less 64. Used as subscript for T.

In this form the program will produce a rather untidy set of numbers. We can improve the output by using the CHR$ function, which does the opposite to ASC: it converts an ASCII code number into the corresponding one-character string. For instance, the command

```
PRINT CHR$(68)
```

gives D.

We can change the last few lines of the program to read

```
100 FOR P = 1 TO 26 STEP 2
110 PRINT CHR$(P+64);T(P),CHR$(P+65);
    T(P+1)
120 NEXT P
130 STOP
```



The program now displays a proper table, 13 lines long, with two entries per line. The example below gives a typical display:

```
SIBELIUS
WAS VERY REBELIUS
WHEN SCORING FOR THE TIMPANI
IN HIS FIRST SYMPANI

A 3    B 2
C 1    D 0
E 6    F 2
G 1    H 3
I 10   J 0
K 0    L 2
M 2    N 5
O 2    P 2
Q 0    R 5
S 8    T 3
U 2    V 1
W 2    X 0
Y 2    Z 0
```

In this program the function CHR$ is used to convert the numerical sequence 1,2,3 . . .26 into the strings "A", "B", "C", . . . "Z".

## USING ASC — IGNORING ILLEGAL INPUT

The second area where ASC is useful is in handling data which, for whatever reason, can't be handled by the normal INPUT command. To give a simple example you may be designing an interface for users so naive — or so clumsy — that they can't be trusted to type a number without hitting all sorts of wrong keys. You would like to make matters easy by getting the machine to ignore all keys except the ten decimal digits 0-9, the DEL key to erase mistakes, and the RETURN to terminate the number. If a key is totally ignored then its character won't even be displayed on the screen, so the user needn't be aware that he has actually hit it.

A suitable specification, flow chart and subroutine are given below. Note that all the meaningful characters including DEL and RETURN are detected by their ASCII codes. The 'DEL' key strips away the right-most character in the string being assembled. It also overwrites the symbol displayed on the screen with a space, and then moves the cursor back, so that the *next* character typed will appear in the right place. Thus screen erasure takes three characters: cursor left, space, cursor left.

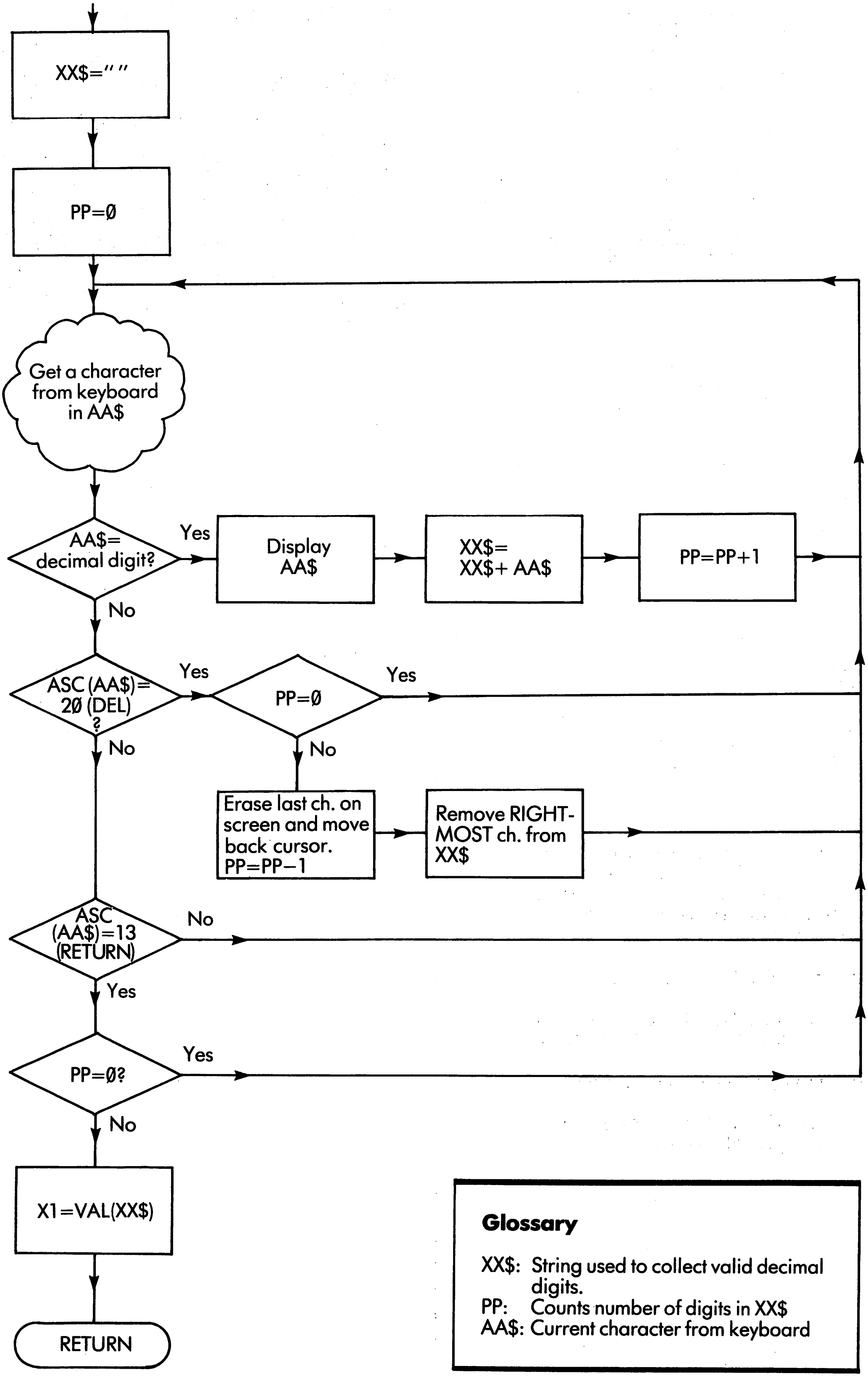
**Subroutine Specification**

*Purpose:* To read a number from the keyboard, ignoring all meaningless characters.

*Lines:* 7000-7090

*Parameters:* *Output:* Result delivered in X1

*Locals:* PP, AA$, XX$




**Glossary**

XX$: String used to collect valid decimal digits.
PP: Counts number of digits in XX$
AA$: Current character from keyboard

```
7ØØØ REM ROBUST NUMBER INPUT
7Ø1Ø XX$ = " ": PP=Ø
7Ø2Ø GET AA$: IF AA$ = " " THEN 7Ø2Ø
7Ø3Ø IF AA$ >= "Ø" AND AA$ <= "9"
     THEN PRINT AA$;: XX$ = XX$ + AA$:
     PP = PP+1 : GOTO 7Ø2Ø
7Ø4Ø IF ASC(AA$) <> 2Ø THEN 7Ø7Ø : REM
     CHECK FOR DEL (=2Ø)
7Ø5Ø IF PP = Ø THEN 7Ø2Ø : REM CAN'T
     DELETE NOTHING
7Ø6Ø PRINT " [SHIFT] and [CRSR ⇐⇒] space
     [SHIFT] and [CRSR ⇐⇒] ;: PP = PP-1 :
     XX$ = LEFT$(XX$,PP):GOTO 7Ø2Ø
7Ø7Ø IF ASC(AA$) <> 13 THEN 7Ø2Ø :REM
     LOOK FOR RETURN
7Ø8Ø IF PP = Ø THEN 7Ø2Ø :REM MUST BE
     SOME DIGITS
7Ø9Ø X1 = VAL(XX$) : RETURN
```

The relationship between CBM ASCII and screen codes is irregular. The five right-most binary digits are always the same: but the other digits don't follow any simple pattern. The situation is expressed in the table below:

| Top 3 bits of CBM ASCII code | Numerical range of CBM ASCII code | Top 3 bits of screen code | Comments |
|---|---|---|---|
| ØØØ | Ø-31 | — | Control characters |
| ØØ1 | 32-63 | xØ1 | |
| Ø1Ø | 64-95 | xØØ | |
| Ø11 | 96-127 | — | Not used |
| 1ØØ | 128-159 | — | Control characters |
| 1Ø1 | 16Ø-191 | x11 | |
| 11Ø | 192-223 | x1Ø | |
| 111 | 224-255 | — | Not used |

In the screen code, x=Ø for a normal character, and x=1 for a reversed character.

The following commands may be used to convert between CBM ASCII code (AA) and screen code (SS):

a) From CBM ASCII to Screen:

```
SS=(AA AND 31) +Ø.5★(AA AND 128):
   IF (AA AND 64)=Ø THEN SS=SS+32
```

b) From Screen code to CBM ASCII:

```
AA=(SS AND 31) + 2★(SS AND 64) -
   (SS AND 32)+64
```

# EXPERIMENT 24·1

Write a program which allows the user to make simple thick-line drawings. Initially, your program displays a black reversed space in the middle of the screen. This is the beginning of a continuous line which is extended one space upward when the user hits the F1 function key. Similarly, the line is extended to the right, downwards or to the left in response to the F3, F5 and F7 keys respectively.

Use your program to draw a spiral.

| Experiment 24.1 Completed | |
|---|---|

## THE "ON" COMMAND

Another facility which is sometimes useful, is the ON command. This command allows the program to jump in any of several directions depending on the value of a variable or expression.

```
ON  A  GOTO  1ØØ,  15Ø,  18Ø,  195,  23Ø
```

keyword

number variable or expression

list of line numbers separated by commas

The VIC takes the value of the variable (or expression) and uses it to select *one* of the label numbers in the list. If the value is 1 it takes the first, if 2 it takes the second, and so on. If the value is less than 1, or higher than the number of labels in the list, there is no jump at all.

The example above is equivalent to:

```
IF A = 1 THEN 1ØØ
IF A = 2 THEN 15Ø
IF A = 3 THEN 18Ø
IF A = 4 THEN 195
IF A = 5 THEN 23Ø
```

If A is less than 1 or greater than 6, the line following the ON will be executed next.

In VIC BASIC, there is also a version of the ON command which uses GOSUB instead of GOTO.

One use of the ON command might be in a program which presented the user with a 'menu' of options, like this:

```
 1Ø PRINT "DO YOU WANT ADVICE ON"
 2Ø PRINT "STORING PROGRAMS (1)"
 3Ø PRINT "USING RND (2)"
 4Ø PRINT "DRAWING PICTURES (3)"
 5Ø PRINT "THE ASCII CODE (4)"
 6Ø PRINT "SOUND PRODUCTION (5)"
 7Ø INPUT "TYPE 1-5";X
 8Ø ON X GOSUB 3ØØ,4ØØ,5ØØ,6ØØ,7ØØ
 9Ø GOTO 1Ø
...
3ØØ REM GIVES ADVICE ON STORING
    PROGRAMS
...
39Ø RETURN
4ØØ REM GIVES ADVICE ON USING RND
...
49Ø RETURN
...
...
7ØØ REM GIVES ADVICE ON SOUND
    PRODUCTION
...
79Ø RETURN
```

## THE "END" COMMAND

Most of the programs in this book have used STOP to return control to the keyboard when a program ends. An alternative command is

END

The difference is that when it is executed, it doesn't say in which line the 'break' occurred; it just comes up with 'READY'. You can use STOP or END as you please.

## THE "DEF" COMMAND

The next feature to be described, the DEF facility, is frankly one of the least useful parts of VIC BASIC. It is suggested that unless you're a good mathematician and specially interested in formulas you skip straight to the next section on the use of cassette tapes.

The DEF keyword allows you to *name* a formula, and then refer to it by name instead of writing it out in full each time. The definition is written in terms of a 'dummy' variable which is replaced by an actual value whenever the formula is used. The name of the formula must consist of the letters FN followed by one or two letters or a letter and a digit:

FNA or FNX or FNQC or FNG1

are all proper formula names.

A formula definition might read:

```
1Ø DEF FNB (X) = 1 + 3.73 ★ X ↑ 2 + 93 / X
```

Function name

Dummy variable

Once a function has been put into a program it can be used by writing its name with a suitable argument. Thus

```
2Ø Q = FNB (77)
```

will serve instead of

```
2Ø Q = 1 + 3.73 ★ 77↑ 2 + 93/77
```

or `3Ø PRINT FNB(S)`

can be used for

```
3Ø PRINT 1 + 3.73 ★ S ↑ 2 + 93/S
```

or again,

```
4Ø ZZ = FNB(P−Q)
```

is now a valid way of writing

```
4Ø ZZ = 1 + 3.73 ★ (P−Q)↑2 + 93/(P−Q)
```

Note that in every case the dummy variable X is replaced by the *argument* of FNB.

DEF suffers from severe restrictions which damage its usefulness. Three of the most important ones are:

a) You can't have more than one dummy variable in a definition, for example a formula including SQR(X↑2 + Y↑2) is not allowed as part of function definition.

b) You can't define string formulas, that is to say there is no FNB$ in VIC BASIC.

c) You can't have a subroutine (as opposed to an expression) to work out your value. You are restricted to a formula even though you might think subroutines make good sense.



# EXPERIMENT 24·2

(For mathematicians only!)

a) Define a function FNA to work out the formula

X↑3 + (X+7)↑2 − 1ØØ

Use it to tabulate the value of $x^3 + (x+7)^2 - 100$ for values of x between 2 and 3, going up in steps of Ø.1. Estimate as well as you can the value of X for which

$x^3 + (x-7)^2 - 100 = 0$

b) Define a second function FNB for the formula

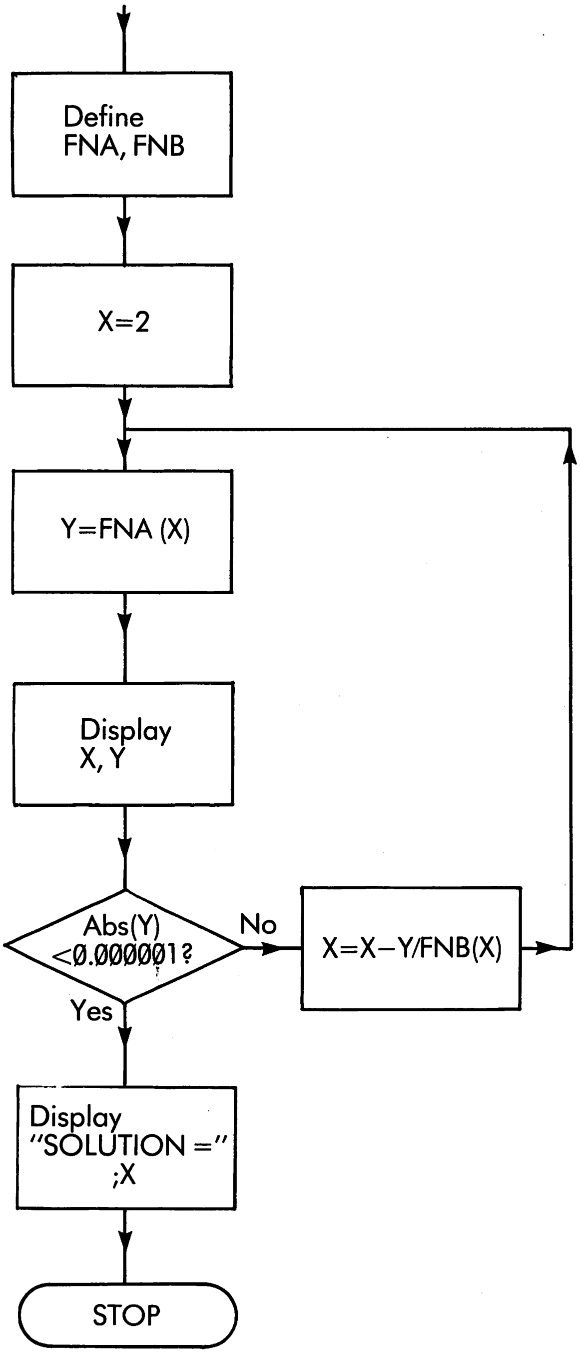
3 ★ X ↑ 2 + 2 ★ (X + 7)

(Calculus fiends note that this is the *derivative* of the first function with respect to X.)

Now write a program to calculate an approximate solution to the equation

$x^3 + (x-7)^2 - 100 = 0$

using the Newton-Raphson method. An appropriate flow chart is:




**Glossary**

X: Current value of x
Y: Current value of $x^3 + (x-7)^2 - 100$

*Experiment 24.2 Completed*

## STORING AND RETRIEVING DATA ON CASSETTE

Non-mathematical readers rejoin here! Next, we look at the commands for storing and retrieving *data* (as opposed to programs) on cassette tapes. For example, you may have a large collection of scientific observations or replies to a questionnaire which you want to analyse with several different programs. Clearly it is worthwhile keeping this data in 'machine-readable' form so that you don't have to type it over and over again.

The basic unit of storage on a cassette tape is the *file*. It is arranged in three sections:

| TRAILER | FILE BODY | HEADER |
|---|---|---|
| end of file marker | characters | file name |

Direction of tape movement →

This diagram shows a segment of tape 'unwound' from the cassette. The HEADER comes first and identifies the file by holding its name. The name can be any string of characters up to a reasonable length, such as "SATELLITE DATA" or "CRICKET CLUB ADDRESSES".

Next comes the FILE BODY. It contains a sequence of characters and can be as long as you like, up to a full (90 minute) cassette. Each minute of playing time holds about 1200 characters.

The file body is divided into equally sized 'blocks' each holding about 256 characters. Each block is followed by a gap which allows the tape mechanism to stop and start between blocks.

Finally the TRAILER holds a special group of symbols which marks the end of the file.

In practice you don't need to know much about the details of these arrangements because the record system works automatically.

## PRINT # — TO WRITE DATA

A single cassette tape can hold several different files recorded one after the other. The only possible snag with this arrangement is that to read the later files you have to get past the earlier ones first.

To write a file on to a cassette we use three new commands:

```
OPEN 1,1,2,"file name"
PRINT#1,
CLOSE1
```

To start a file, your program must give an OPEN command in the form shown above. The file name can be selected freely, but the numbers 1,1 and 2 must be precisely as they are shown.

When the OPEN is executed, the VIC comes up with a message on the screen saying

PRESS RECORD AND PLAY ON TAPE

Load a blank cassette into the player (or if not blank, then one which has been wound past the files or programs you still need) and press the correct control keys. The tape must be long enough for the file you want to write, since there is no way of changing tapes in mid-file. Remember to press RECORD so that it stays down. If you forget the VIC will go through all the motions of writing a file but won't actually put any information on the tape. Don't be caught out!

Once the tape is loaded, the VIC will write a header with your file name. Then you can start sending data to it with the command

```
PRINT#1,
```



Note that all the 8 characters in this keyword are inseparable and must be typed exactly as shown. In particular the comma is essential, and you *can't* use ? instead of PRINT.

The keyword should be followed by the names of the variables you want to write. If there is more than one in the command, the names should be separated by the sequence ;",";. The variables may be numbers, strings or a mixture. Some examples are:

```
PRINT#1,X
PRINT#1,P$
PRINT#1,Q$(J);",";X(J);",";R$(J+1)
```

You can have as many variables as you like in the PRINT#1, command, provided that

a) The length of the *command* doesn't exceed 88 characters (this is the normal limit which applies to all commands).

b) The total number of *characters* sent to the tape by any one command is less than 8Ø.

If you break the second rule nothing will seem to go wrong when you write the data, but you won't be able to read it back later. Beware!

You can, of course, use PRINT#1, repeatedly inside a loop to write all the information you like.

If you are writing more than a very few variables you'll notice that the tape moves in jerks. This is because the VIC has an internal reservoir of information which acts as a 'buffer' between your program and the tape. As the machine executes PRINT#1, commands, the data you use is first collected in the buffer. When the buffer is full, its contents are sent out to the cassette in a single burst to write a block. Then the tape stops, the buffer is cleared and the process starts all over again.

When you've written all the information you want to record, give a CLOSE1 command. This forces the VIC to write another block (even though the buffer may only be part-full) and a trailer with the end-of-file marker.

## INPUT # TO READ DATA

To get your information back from a cassette tape, you need the three instructions

```
OPEN 1,1,Ø,"FILE NAME"
INPUT#1,
CLOSE1
```

The OPEN command with the zero in front of the file name (instead of a 2) makes the VIC open a file for *reading* only. The machine displays the message

```
PRESS PLAY ON TAPE
```

and waits for you to put your cassette in the player and press PLAY. *Don't* press RECORD if you value your data.

When it senses that the tape is loaded, the VIC begins to search the tape for a file with a name which matches the one in the OPEN command. The matching process only requires that the string in the OPEN command should be the same as the *beginning* of the file name. If the actual file name is "THURSDAY DATA" the file will be opened by any of the following commands

```
   OPEN 1,1,Ø,"THURSDAY DATA"
or OPEN 1,1,Ø,"THURSDAY"
or OPEN 1,1,Ø,"T"
or OPEN 1,1,Ø,"" : REM NULL STRING WILL
          OPEN ANY FILE
or OPEN 1,1,Ø : REM TITLE CAN BE OMITTED
or X$ = "THURS" : OPEN 1,1,Ø,X$ : REM
          VARIABLE CAN BE USED
```

If the title is given as a null string or omitted, the command will open the first file it comes to, no matter what it is called.

The INPUT # 1, command is like the PRINT # 1, in reverse. The keyword is followed by the names of the variables to be read from the tape, separated by commas if there is more than one. Examples are

```
INPUT # 1,Z
INPUT # 1,P$
INPUT # 1,R$,Q,T$
```

Note that the number and type of the variables which follow INPUT # 1, must be the same as that used to put the values on the tape in the first place. The command

```
INPUT # 1,A$,B$,C :REM TWO STRINGS
AND A NUMBER
```

could be used to fetch data originally written by

```
   PRINT # 1,A$;",";B$;",";C :REM TWO
          STRINGS AND A NUMBER
or PRINT # 1,Z$(Q);",";"LEADS TO";",";
          X :REM TWO STRINGS AND A
          NUMBER
```

but if the data had been written as Y;",";P$ the above INPUT would not work.

When the system is reading from a cassette tape various unexpected things can happen. To allow for this difficulty the VIC reserves a special variable called ST (for "STatus") and uses it to return a coded report every time the input # 1, command is obeyed. The value Ø means that all is well. 64 signals that you have reached the end of the file, and other values imply that something has gone wrong: the tape has become corrupted, or perhaps it wasn't properly recorded in the first place.

To illustrate the action of the cassette deck, here is a pair of programs. The first one lets you draw a picture on the screen using the cursor and colour controls, and then records this picture on a file by PEEKing values off the screen and colour RAMs and writing them as numbers. The second program reads down the file and reconstructs the picture. Study both programs carefully, and notice the way ST is used. Then key them in one by one, and try them out.

```
10 OPEN 1,1,2,"SCREEN PICTURE"
20 PRINT" [SHIFT] and [CLR HOME] [CRSR ↑↓]
   DRAW ANY PICTURE YOU"
30 PRINT"LIKE, USING THE CURSOR"
40 PRINT"AND COLOUR CONTROLS."
50 PRINT"LEAVE THE CURSOR ON"
60 PRINT"THE TOP LINE,WHICH
70 PRINT"MUST BE EMPTY."
80 PRINT"THEN PRESS RETURN"
85 FOR S = 1 TO 5ØØØ : NEXT S
90 INPUT" [SHIFT] and [CLR HOME] ";X$ :
   REM USER DRAWS PICTURE

100 FOR J = Ø TO 5Ø5 : REM SCAN SCREEN
    AND COLOUR RAMS
110 PRINT # 1, PEEK(768Ø+J);",";PEEK
    (384ØØ+J)
120 NEXT J
130 CLOSE1
140 STOP
```

now rewind the tape and type in the following program:

```
10 OPEN 1,1,Ø,"SCREEN PICTURE"
20 J=Ø
30 INPUT # 1,X,Y : REM GET SCREEN AND
   COLOUR CODES
40 IF ST <> Ø THEN 7Ø
50 POKE 768Ø+J,X : POKE 384ØØ+J,Y
60 J = J+1 : GOTO 3Ø
70 IF ST = 64 THEN 1ØØ : REM CHECK FOR
   END OF FILE
80 PRINT "TAPE FAULT"
90 STOP

100 CLOSE
110 GOTO 11Ø : REM LOOP STOP IF ALL
    WELL
```

**GET #**

Another command which is sometimes used with cassette tapes is

GET # 1,

This command is rather like PRINT # 1, except that it transfers single characters. It would appear in sequences like

```
...
1ØØ GET A$ : IF A$ = " " THEN 1ØØ : REM
    GET A CH. FROM KEYBOARD
11Ø PRINT A$:REM DISPLAY IT
12Ø PRINT # :1, A$:= REM SEND IT TO
    CASSETTE TAPE
...
```

and

```
...
2ØØ GET # 1,X$: REM GET A CHARACTER
    FROM TAPE
21Ø IF ST <> Ø THEN 3ØØ :REM JUMP IF END
    OF FILE OR ERROR
22Ø PRINT X$;
...
```

**PROBLEMS**

As you have seen, the VIC gives you the basic facilities for writing data to a cassette and bringing it back later. These facilities are primitive and have certain drawbacks:

1) Reading and writing is s l o w.

2) The reliability of the cassette system is not perfect. Cassettes can be flawed even when they are bought, or they can be damaged by damp, rough handling, excessive heat or cold, or strong magnetic fields. All these circumstances may produce errors in your files. The error rate when storing data is much higher than you get with programs, because

   a) Data files are generally longer

   b) There is no way of 'verifying' data files like programs

   c) The SAVE command in the VIC actually records each program twice, on different parts of the tape. The system can therefore have two tries at reading the program back correctly. Data, on the other hand, is only recorded once, and a single error is fatal.

3) The VIC can only handle one cassette. This means that you cannot edit a file or add data to it unless it is short enough to fit wholly into the VIC's memory.

If you are seriously interested in storing large amounts of data, you should invest in a floppy disk drive. Commodore have a very good unit specifically designed for the VIC. If you do decide to go ahead with cassettes, use the best quality tape you can get, keep your cassette player clean and in perfect condition, and be prepared for occasional disappointments.

# EXPERIMENT 24·3

In this experiment we shall design and build a mechanical Morse Code teacher.

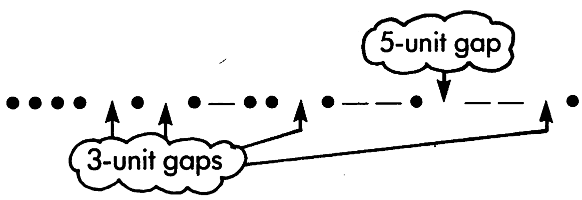
Morse is used to send information by radio. Each letter of the alphabet has a code which consists of dots and dashes. The full code is

| | | | | | |
|---|---|---|---|---|---|
| A | •— | J | •——— | S | ••• |
| B | —••• | K | —•— | T | — |
| C | —•—• | L | •—•• | U | ••— |
| D | —•• | M | —— | V | •••— |
| E | • | N | —• | W | •—— |
| F | ••—• | O | ——— | X | —••— |
| G | ——• | P | •——• | Y | —•—— |
| H | •••• | Q | ——•— | Z | ——•• |
| I | •• | R | •—• | | |

The basic time unit is the dot, and other time intervals are defined as follows:

| | |
|---|---|
| Dash | 3 dots |
| Gap between dots and dashes in the same letter | 1 dot |
| Gap between letters | 3 dots |
| Gap between words | 5 dots |

For instance, the message HELP ME would be sent as




In Morse, punctuation is ignored.

To begin, write a subroutine which takes two parameters:

a) A string containing a sentence

b) A number giving the desired speed of transmission in 1/1000'ths of a second per dot.

The subroutine converts the sentence into Morse and transmits it on the VIC sound generator at the required speed.

HINT: Set up a subroutine which emits a sound — or a silence — of a given length. Drive it with a program which uses a two-dimensional table like this:

| | | | | | |
|---|---|---|---|---|---|
| 1 | 3 | Ø | Ø | Ø | (A) |
| 3 | 1 | 1 | 1 | Ø | (B) |
| 3 | 1 | 3 | 1 | Ø | (C) |
| 3 | 3 | 1 | 1 | Ø | (Z) |

The table contains the code for each of the letters A-Z, and should be set up using READ and DATA statements.

When you are satisfied that your subroutine is correct, move on to the second part of the experiment. This involves two programs:

a) A program to input a text (a whole series of English sentences) from the keyboard and record them on a cassette file. Use the dummy sentence "ZZZZ" to end your input.

b) A program to read back the sentences and transmit them in Morse code at any desired speed.

When the programs are working, they can be used to prerecord messages and transmit them at very high speed, so saving time and increasing the capacity of a radio channel. They are also helpful if you want to practise receiving Morse, since you can start slowly and gradually work up your speed. For best results, you should get someone else to tape the sentences to be transmitted, so you don't know what to expect in advance.

| *Experiment 24.3 Completed* | |
|---|---|

# EXPERIMENT 24•4



This experiment invites you to design and write a Computer Dating program.

Load the program "MAKENAMES" from the cassette tape. Then put a fresh cassette into the recorder and run the program. It will produce a list of 1ØØ people and write their personal details on to the cassette in a file called "COMPUTER DATES".

The record for each person consists of the following items (in the order they are recorded):

NAME (string)

ADDRESS (string)

TOWN (string). One of EDINBURGH, GLASGOW,DUNDEE,or ABERDEEN.

SEX (string). One of F or M.

AGE (number)

HEIGHT (number). Height in inches.

MAIN HOBBY (string)
SECOND HOBBY (string)
} Both taken from the following list: FOOTBALL,TENNIS, HILL-WALKING, OPERA, JAZZ, ROCK, THEATRE, READING, POLITICS, STUDYING, CHESS, GAMBLING,HORSE-RACING,CARS,MOTOR-BIKES,CYCLING,MEETING PEOPLE

POLITICS (string) One of CONSERVATIVE, LABOUR,LIBERAL,SDP, OTHER,NONE

When you have got the file of people, begin by writing the simplest program which opens the file, reads down and displays the records one at a time. Next, design and write a 'dating' program which asks for the personal particulars of a 'customer', and then searches the file and picks out the most suitable 'date'. Assume that the person selected must live in the same town. People who satisfy these constraints score 'bonus' points on the following (arbitrary) scale:

| | |
|---|---|
| Age compatibility: If the girl is the same age or not more than 4 years younger than the man: | 3 points |
| Height compatibility: If the girl is the same height or not more than 4 inches smaller than the man: | 2 points |
| Hobbies: For each shared hobby: | 5 points |
| Politics: For a common political viewpoint: | 3 points |
| If one supports Labour and the other Conservative: | *minus* 4 points |



The 'best' match is the one with the highest score.

*Warning: If you find someone you really like, don't bother to contact them: the people on the file aren't real.*

| *Experiment 24.4 Completed* | |
|---|---|

# UNIT: 25

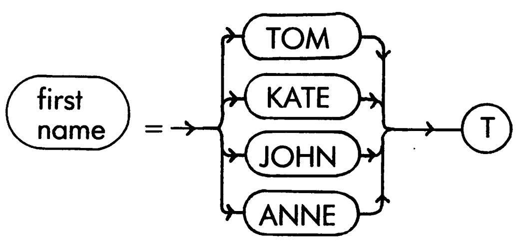
## PROGRAM DESIGN — CASE STUDIES

The topic of program design has been a recurring theme throughout this book. This final Unit consists of some case studies, each of which shows how an apparently complex problem may be solved quickly and easily by examining its structure and transferring this structure to the program itself.

## RANDOM SENTENCES — TRAMLINE GRAMMAR

The first of our case studies is concerned with the production of random sentences, like those in Unit 6. Clearly, these sentences cannot be mere collections of words strung together in any order; if they are to make sense they must follow the rules of grammar.

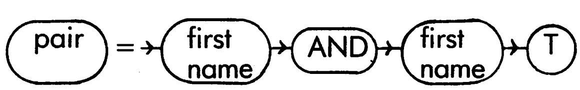
The notation often used for these rules is called a "tramline" grammar. Suppose that at a certain point in a sentence being constructed we need a person's name to be selected from the list TOM, KATE, JOHN, ANNE. We can write down this part of the grammar with the aid of the diagram




Imagine a tram entering the diagram from the left (in the direction of the arrow). When the driver comes to a junction, the direction he takes is decided at random. The tram is eventually certain to arrive at the terminus on the right, but it may do so by any of four routes: TOM, KATE, JOHN or ANNE.

In this diagram, each oval contains a word which is a possible candidate for part of the sentence being constructed. Ovals in tramline diagrams may also hold the names of other diagrams. The difference is always clear because the names of diagrams are written in small letters.

Look at the following tramline diagram:

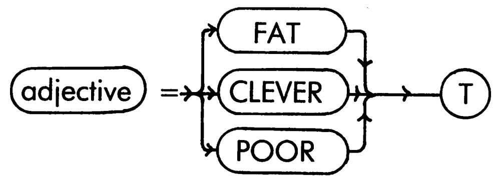



(where "first name" is the diagram with TOM, KATE, JOHN and ANNE).

For the tram-driver, first name is a kind of subroutine. By the time the tram has found its way across the pair diagram, it may have come up with any of the following phrases:

TOM AND JOHN
ANNE AND TOM
or even KATE AND KATE

If you want to define a phrase with a variable number of words, you can put a branch in the diagram. If you define



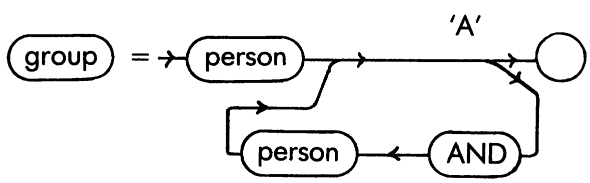

then the grammar


will yield

POOR JOHN
FAT ANNE
KATE
or POOR TOM

because the tram driver can choose, at random, whether to go down the adjective route or not.

Tramline grammars may have loops in them. Consider the diagram




Remember that the driver has a free choice when he gets to point A in the diagram. If he goes straight on he'll reach the terminus and end the phrase. If he turns right he'll add another person. Some of the phrases he might produce are

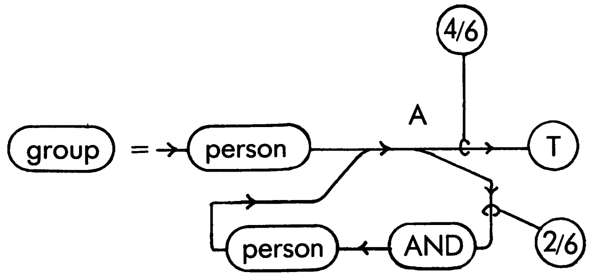
TOM
CLEVER TOM AND KATE
FAT TOM AND POOR ANNE AND CLEVER JOHN
TOM AND JOHN AND POOR KATE
or possibly FAT TOM AND FAT TOM AND FAT TOM

In some ways the tramline diagram is like a flow chart. The vital difference is that at junction points like A, the choice of track is made at random, not in answer to some specific question. The random element is essential otherwise the diagram would produce the same phrase every time.

## PROBABILITY

Although the selection of route is not determined in advance, we'd still like to keep some control over it; otherwise the driver could decide to keep on going round the loop for ever. We can do this by attaching a *probability* or *likelihood* to each of the possible tracks.

One way of doing this is to give the driver a six-sided die, and instruct him to toss it whenever he has to make a choice. At point A, for example, we tell him to turn right if he throws a 5 or a 6, but to go on to the terminus for a 1,2,3 or 4. This means that in the long run he can expect to turn right twice out of every six times he passes A, and to go on four times. We can indicate this on the diagram by attaching *probability markers* like this:




Note that the probabilities of the routes stemming from one point such as A must add up to certainty, because the tram driver is *bound* to take one of them. If you regard the probability markers as fractions, they have to add up to 1. In the definition of (group) they do: 4/6 +2/6 = 6/6 = 1.

Now we'll consider how the VIC can be made to produce random phrases.

The ground rules are as follows:

1) The phrase being built is kept in the string variable X1$. The variable starts as a null string, and words are attached to it one at a time. Each word is preceded by a space.

2) Every separate grammar diagram is represented by a *subroutine*. One of its parameters, for both input and output, is X1$.

Let's look at some elementary operations. To attach a known word to the phrase we simply concatenate it, like this:

```
X1$ = X1$ + " AND"
```

note preceding space

To choose a random word from a list the simplest method is to ensure that all the candidate words are in an array. Suppose there are J of them in consecutive elements starting at N$(K) and ending at N$(K+J−1). Then the following command will pick one at random and attach it to X1$:

```
X1$=X1$+" "+N$(K+INT(J★RND(Ø)))
```

This works because the subscript expression K+INT(J★RND(Ø)) is equally likely to come up with any number between K and K+J−1: exactly what we need.

To make a tramline route with a given probability we can use the condition

```
RND(Ø) < p/q
```
(where p/q is the probability marker)

If we put

```
RND(Ø) < 4/6
```

the condition will be true four times out of six, and false the other two times. This means that the other probability marker (2/6) doesn't have to be written down in your program.



We can now construct a program to produce group phrases as they are defined by our grammar. We begin by setting up an array with names and adjectives, and we initialise X1$ to be null. This occupies lines 1Ø-4Ø in the program below.

Next, we write a subroutine for each of the tramline diagrams. The one starting at line 1ØØØ is for a (first name). The constants in the subscript expression are *1* and *4* because there are *4* possible names starting at N$(*1*). Similarly, the subroutine which starts at 11ØØ produces an (adjective) and the appropriate constants in the subscript expressions are 5 and 3.

The subroutine at 12ØØ yields a (person). We make the probability of using an (adjective) 3/6 — which implies that the probability of *not* having one is also 3/6: even chances.

At 13ØØ you'll find the subroutine for the (group) diagram. Notice how precisely it follows the tramlines:

1310 selects the leading person

132Ø makes a random decision whether to end the phrase

133Ø puts in the word AND

134Ø selects another person

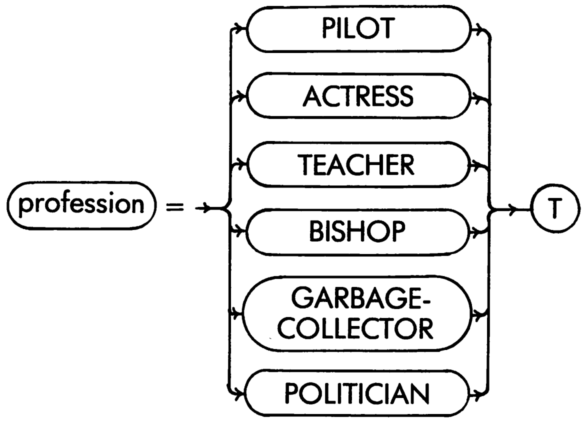
1350 returns the subroutine to the point where it decides whether to stop or go round again.

Finally, we supply some "driver" commands in lines 4Ø, 5Ø and 6Ø.

```
  1Ø DIM N$(7)
  2Ø N$(1) = "TOM": N$(2)="KATE":
     N$(3)="JOHN":N$(4)="ANNE"
  3Ø N$(5) = "FAT": N$(6)="CLEVER":
     N$(7)="POOR"
  4Ø X1$=" "
  5Ø GOSUB 13ØØ
  6Ø PRINT X1$
  7Ø GOTO 4Ø

1ØØØ REM FIRST NAME
1Ø1Ø X1$=X1$+" "+N$(1+INT(4★RND(Ø)))
1Ø2Ø RETURN

11ØØ REM ADJECTIVE
111Ø X1$=X1$+" "+N$(5+INT(3★RND(Ø)))
112Ø RETURN

12ØØ REM PERSON
121Ø IF RND(Ø) < 3/6 THEN GOSUB 11ØØ:
     REM CALL ADJECTIVE
122Ø GOSUB 1ØØØ : REM CALL PERSON
123Ø RETURN

13ØØ REM GROUP
131Ø GOSUB 12ØØ : REM CALL PERSON
132Ø IF RND(Ø) < 4/6 THEN RETURN : REM
     POINT "A" IN DIAGRAM
133Ø X1$=X1$+" AND"
134Ø GOSUB 12ØØ : REM CALL ANOTHER
     PERSON
135Ø GOTO 132Ø
```

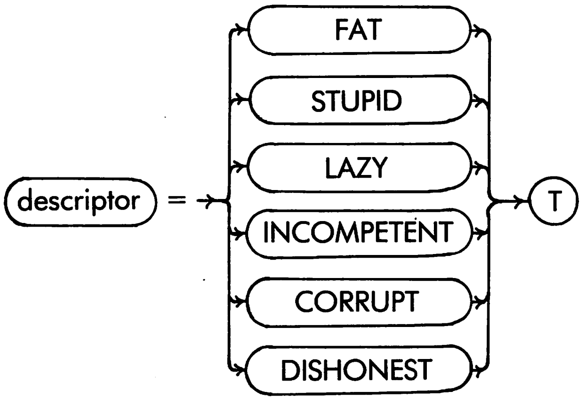
Key in this program and run it. Then try the effect of varying the probability markers in lines 121Ø and 132Ø.

Once the principles of building random phrases have been established, they can easily be extended to complete sentences. Study the following definitions, and write down examples of the phrases or sentences they might produce:



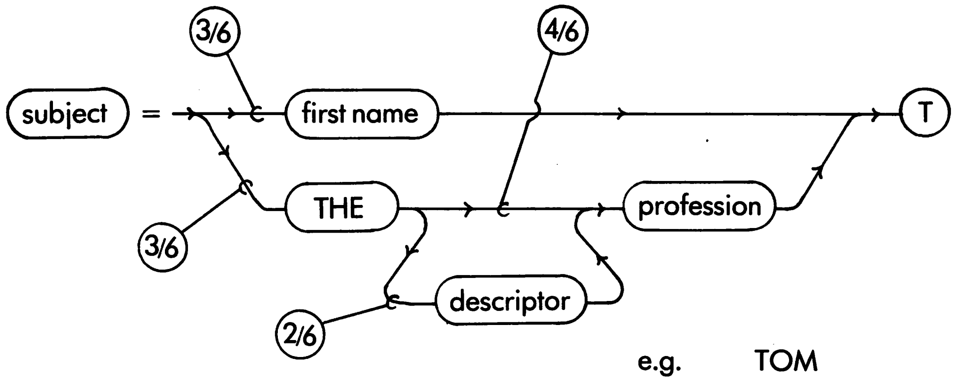

e.g. BISHOP

or ............




e.g. LAZY

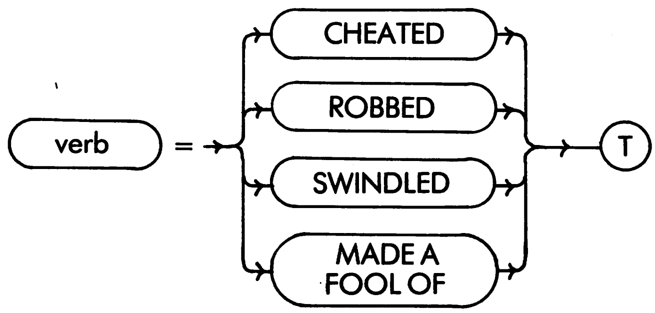
or ............




e.g. TOM

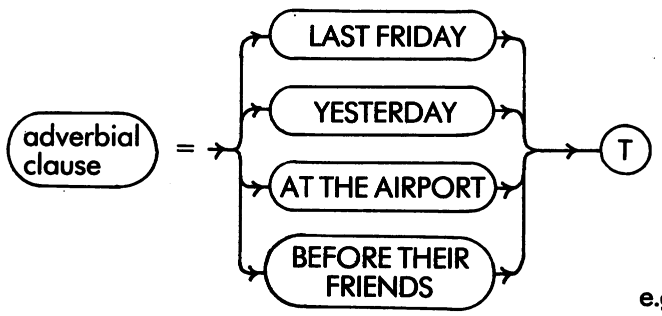
or THE STUPID PILOT

or ...............



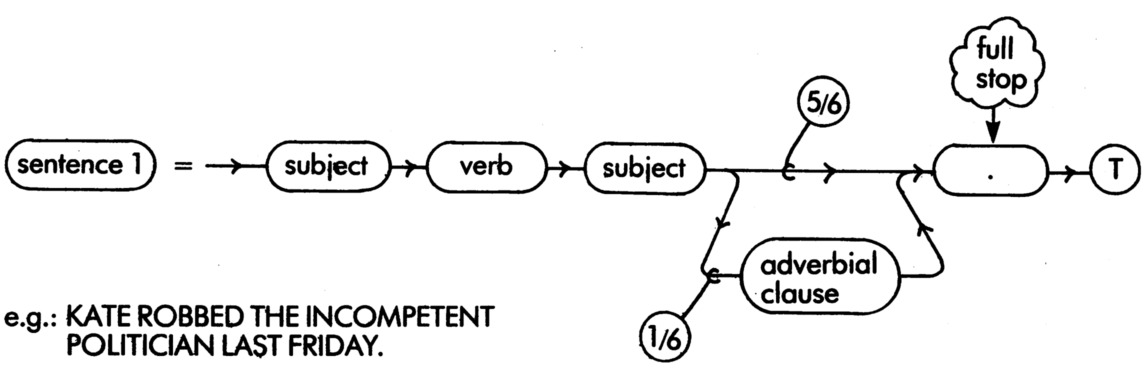

e.g. ROBBED

or .........




e.g. YESTERDAY

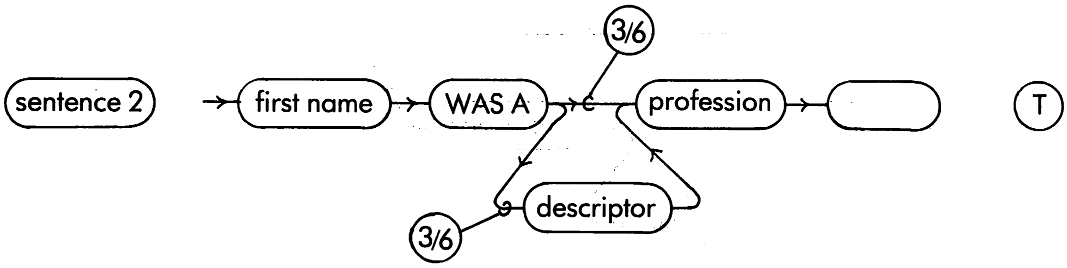
or ...........




e.g.: KATE ROBBED THE INCOMPETENT POLITICIAN LAST FRIDAY.

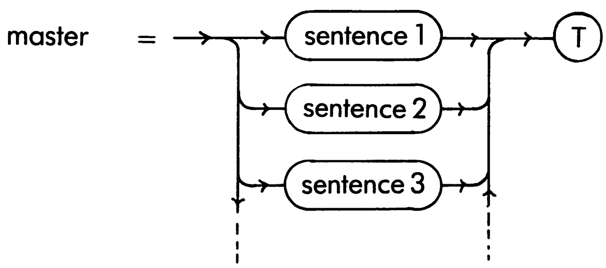
or ......................................

You can have any number of different sentence definitions, such as:




and you can combine them into a 'master tramline diagram which includes all the sentence forms you want to generate. It would begin:

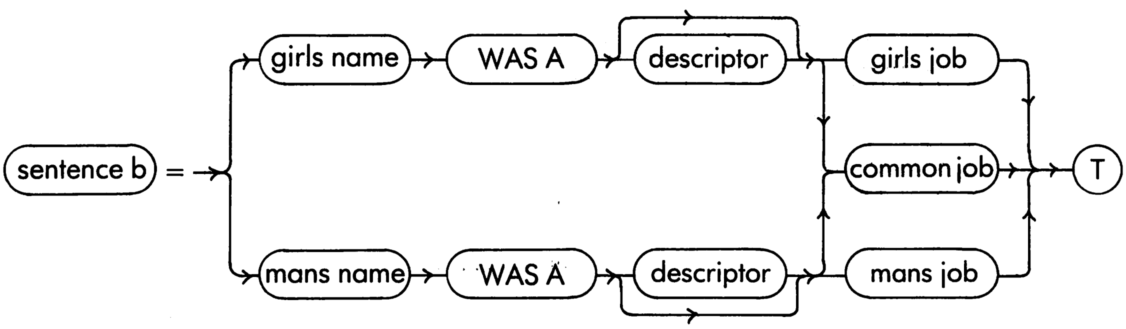
267




At this point, it is as well to watch your grouping of words. Under the present set of tramline diagrams one possible sentence is

TOM WAS A LAZY ACTRESS.

To avoid this kind of absurdity you'd have to separate the names into two groups, and the professions into three: those limited to men (such as BISHOP), those restricted to women (e.g. ACTRESS) and those open to both. An alternative definition for sentence 2 would be:




To complete this case study we should consider some practical details. First, the sentences produced by the system should be properly laid out, and this can be done by the subroutine described in Unit 21.

Second, it makes the program far more interesting for users if they can supply their own lists of words for the various categories — perhaps by altering DATA statements in the program after it has been loaded from a cassette. This implies that the programmer knows neither the words themselves nor how many there will be in each group. There will clearly be some difficulty in writing suitable subscript expressions.

This problem can be overcome by getting the program to set up a set of 'signposts' to the various groups of words.

In our present grammar we have 6 groups of words: first names, adjectives, professions, descriptors, verbs and adverbial clauses. We tell the user to put his choice for each group into one (or more) DATA statements, and to terminate it with a "Z". The user might put:

```
10 DATA RONALD,JIMMY,GERALD,
        RICHARD,Z
20 DATA HANDSOME,BRILLIANT,SUAVE,Z
30 DATA POLITICIAN,PROGRAMMER,
        PERCUSSION-PLAYER,
        COMPUTER SALESMAN,
        CONFIDENCE TRICKSTER,Z
40 DATA WICKED,PROFESSIONAL,
        PRACTICED,Z
50 DATA MET,HELPED,SHOOK HANDS
        WITH,Z
60 DATA LAST NIGHT,IN THE PARK,
        BEFORE HE WAS FIRED,Z
```

Inside the program we arrange the data into arrays: one with an element for each *word* (except the Z's), and one with information about each *group*. The necessary information includes

a) The starting position (that is, the subscript of the first element) in the group

b) the number of words in the group.

A diagram might help to make this clear:

| | |
|---|---|
| 1) | RONALD |
| 2) | JIMMY |
| 3) | GERALD |
| 4) | RICHARD |
| 5) | HANDSOME |
| 6) | BRILLIANT |
| 7) | SUAVE |
| 8) | POLITICIAN |
| 9) | PROGRAMMER |
| 10) | PERCUSSION-PLAYER |
| 11) | COMPUTER SALESMAN |
| 12) | CONFIDENCE TRICKSTER |
| 13) | WICKED |
| 14) | PROFESSIONAL |
| 15) | PRACTICED |
| 16) | MET |
| 17) | HELPED |
| 18) | SHOOK HANDS WITH |
| 19) | LAST NIGHT |
| 20) | IN THE PARK |
| 21) | BEFORE HE WAS FIRED |

N1$

| | | |
|---|---|---|
| 1) | 1 | 4 |
| 2) | 5 | 3 |
| 3) | 8 | 5 |
| 4) | 13 | 3 |
| 5) | 16 | 3 |
| 6) | 19 | 3 |

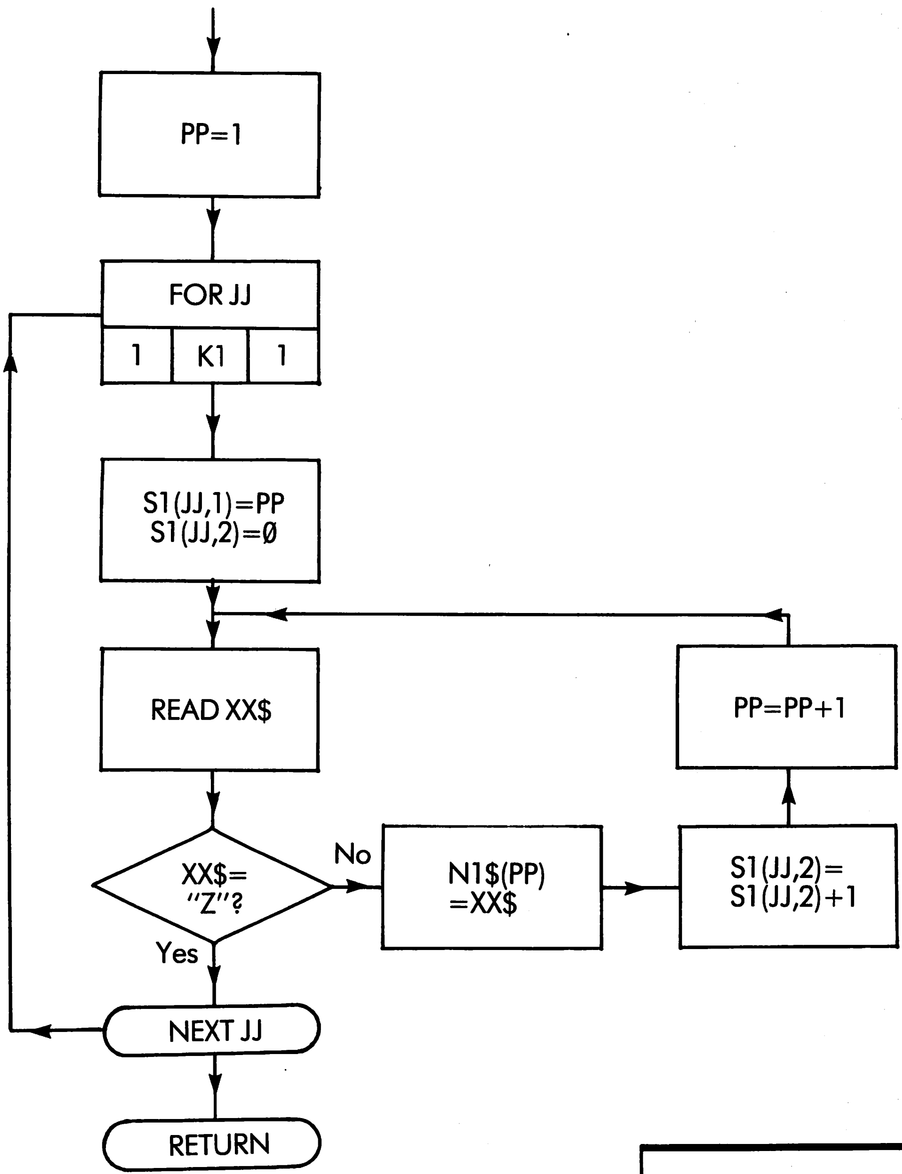
S1

In this data structure, row 3) of array S1 tells the program that group 3 contains 5 words starting at N1$(8). An instruction to add a group 3 word to X1$ would read:

$$X1\$=X1\$+\text{" "}+N1\$(\underbrace{S1(3,1)}_{=8}+INT(\underbrace{S1(3,2)}_{=5}\star RND(\emptyset)))$$

Provided that the signposts in S1 are correctly set up, this expression will work for any collection of words the user supplies.

The setting up of the signposts is illustrated in the following flow chart:




**Glossary**

N1$: Array for *words*
S1: Array for *signposts*
K1: Number of groups
XX$: Current word
JJ: Group counter
PP: Word counter

The corresponding specification and code are:

**Subroutine Specification**

*Purpose:* To read in groups of words for generating random sentences

*Lines:* 5000-5070

*Parameters:* *Output:* N1$(words),S1(signposts)

Empty arrays; K1: Number of groups

*Output:* N1$,S1 (Set up as described in the text)

*Locals:* PP,JJ,XX$

```
5000 REM READ WORDS AND SET UP
     SIGNPOSTS
5010 PP=1
5020 FOR JJ = 1 TO K1
5030 S1(JJ,1)=PP: S1(JJ,2) = 0
5040 READ XX$
5050 IF XX$<>"Z"THEN N1$(PP)=XX$:
     S1(JJ,2)=S1(JJ,2)+1:PP=PP+1:
     GOTO 5040
5060 NEXT JJ
5070 RETURN
```

To summarise: the sentence-generating program we have discussed consists mainly of subroutines which are closely modelled on the grammar of the sentences to be constructed. In this way the structure of the problem is transferred, almost without alteration, to the program itself.

# EXPERIMENT 25·1



Write a complete sentence generator involving several different kinds of sentence. Try it out on your relatives and friends.

| *Experiment 25.1 Completed* | |
|---|---|

## ADVENTURE OR MAZE GAMES

In the next case study we'll take two examples in which the structure of the problem is reflected in the data rather than in the program itself.

There exists a large number of computer games in which you (the hero) have to explore a castle/maze/universe, cope with various dangers such as dragons or aliens, and rescue a princess/hyper-atomic modulator/intrepid space explorer. Commodore have an excellent range of these games in their Adventure series.

The design of a very simple version of one such game can be written down as a diagram, shown in Figure 25.2. When you start coding such a game, the natural way is to begin at the beginning and write reams of shapeless code, like this:

```
10 PRINT " [SHIFT] and [CLR HOME] ";
20 PRINT "YOUR MISSION IS TO RESCUE"

...

100 PRINT" OR B) WAIT AND OBSERVE?"
110 INPUT X$
120 IF X$ <> "A" AND X$ <> "B" THEN
    PRINT "TRY AGAIN": GOTO 110
130 IF X$ = "A" THEN 500
140 PRINT " [SHIFT] and [CLR HOME] ";
150 PRINT "YOU ARE ATTACKED BY"
...

...
270 PRINT "OR B) ESCAPE IN YOUR
    LIFEBOAT?"
280 INPUT X$
290 IF X$ <>"A" AND X$ <> "B" THEN
    PRINT "TRY AGAIN": GOTO 280
300 IF X$ = "A" THEN 500

...
```

and so on.

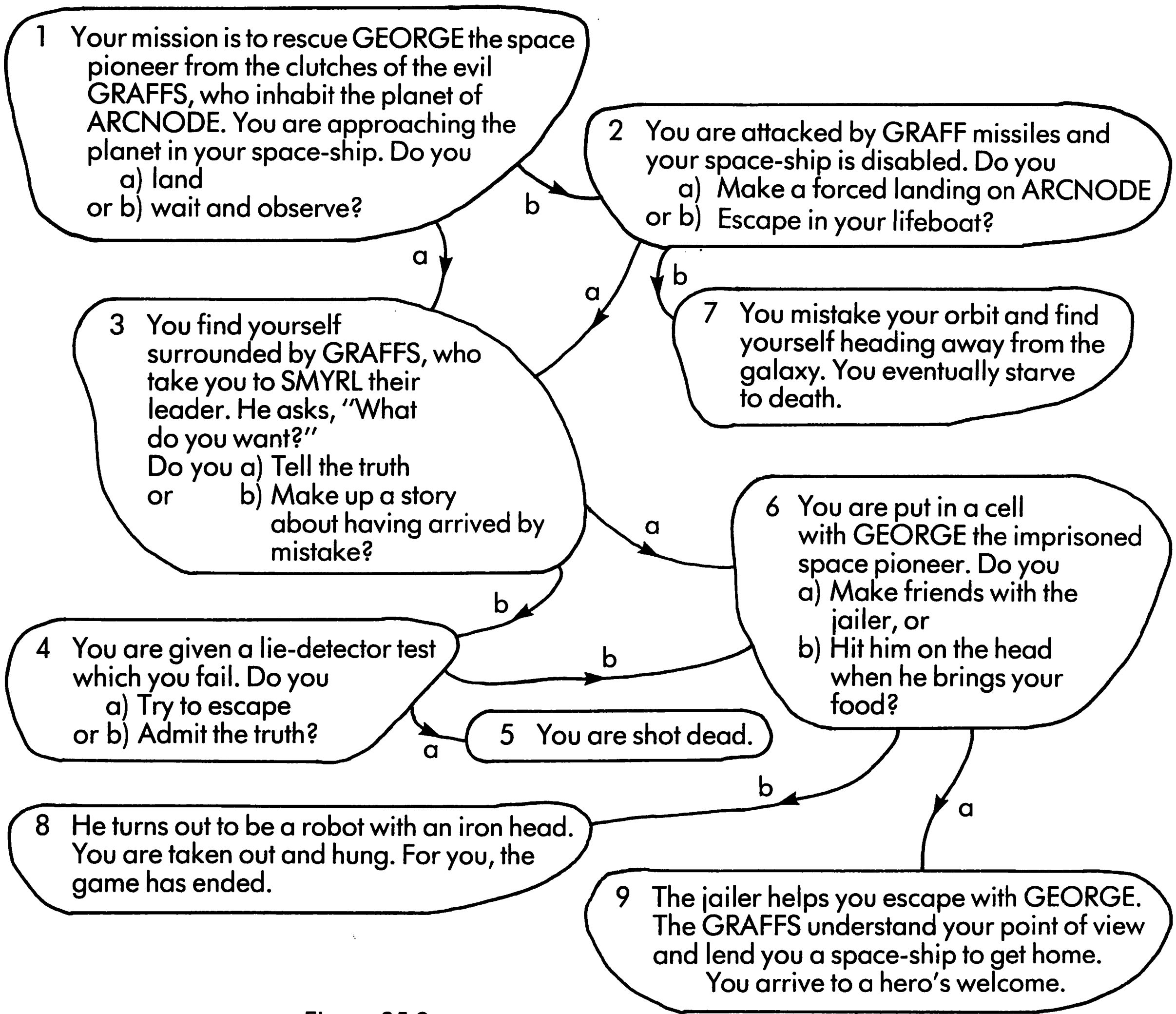



*Figure 25.2*

This program is hard to read and to alter, and will rapidly grow to take up all the space in the memory.

Let's have a look at what happens at each place or *node* in the diagram. There are two possibilities:

1) The computer clears the screen and displays a message. Then it stops the program. This happens when the hero is killed or when he succeeds in his mission.

2) a) The computer clears the screen and displays a message.

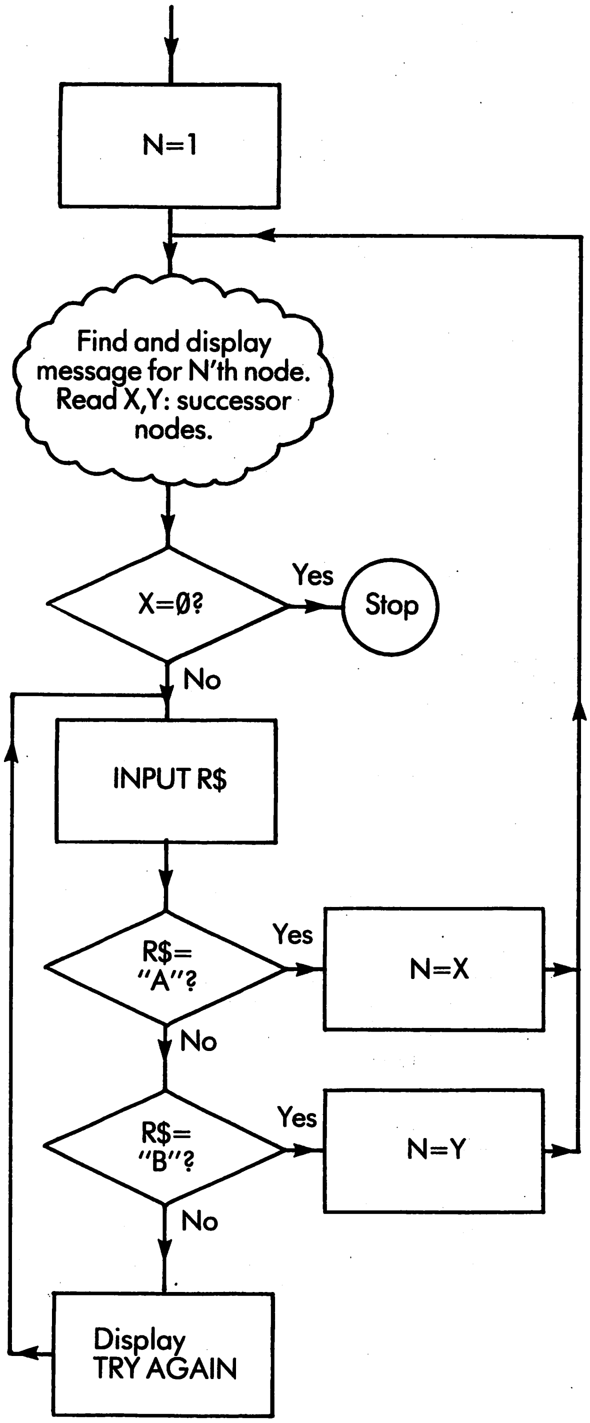
   b) It then invites the user to type A or B, and rejects all other inputs.

   c) It uses the reply to select a new *successor* node, and repeats the process all over again.

This suggests that the game can be administered by a very simple program, with all the complexity about the story-line held in the data.

Suppose we number the nodes from 1 up (as has already been done in the diagram). Then for each node we can make a 'package' of data consisting of the string to be displayed and the numbers of the two successor nodes. We use the convention that a successor number of Ø means that the game has ended.

The basic flow-chart for the program is now quite short. It is:






**Glossary**

N: Current node number
X,Y: Successor node numbers
R$: Reply.

The code for the program is given below. Note that the message for each node is usually too long to go as a single data item. We use four items, which allows for a 'script' of up to 24Ø characters per node. Of the items, the first two describe the new situation and the others give the probable courses of action.

The subroutine at 55ØØ is the one given in Unit 21, to display sentences without breaking up words.

```
10 DATA " [SHIFT] and [CLR HOME] YOUR
    MISSION IS TO RESCUE GEORGE
    THE SPACE PIONEER FROM THE
    CLUTCHES OF THE "
11 DATA " GRAFFS, WHO INHABIT THE
    PLANET ARCNODE, DO YOU"
12 DATA " A) LAND","OR B) WAIT AND
    WATCH",3,2
...
```

followed by similar groups for each of the other nodes. Each group contains 4 strings and 2 numbers.

```
1ØØØ REM PROGRAM BEGINS HERE
1Ø1Ø N=1 : REM START AT NODE 1
1Ø2Ø RESTORE : REM FIND N'TH NODE
1Ø3Ø FOR J = 1 TO N
1Ø4Ø READ J$,K$,L$,M$,X,Y : REM AND
     READ ITS CONTENTS
1Ø5Ø NEXT J
1Ø6Ø X1$=J$ + K$ : GOSUB 55ØØ: REM
     DISPLAY MESSAGE
1Ø7Ø X1$ = L$ : GOSUB 55ØØ :REM FIRST
     ALTERNATIVE
1Ø8Ø X1$ = M$ : GOSUB 55ØØ :REM
     SECOND ALTERNATIVE
1Ø9Ø IF X = Ø THEN STOP : REM END OF
     GAME
11ØØ INPUT R$ : REM GET REPLY
111Ø IF R$ = "A" THEN N=X : GOTO 1Ø2Ø
112Ø IF R$ = "B" THEN N=Y : GOTO 1Ø2Ø
113Ø PRINT "TRY AGAIN": GOTO 11ØØ
```

followed by the commands of the subroutine at 55ØØ.

# EXPERIMENT 25·2

Load the full program of GRAFFS from the cassette tape and try it out for yourself. Then modify it:

a) To produce interesting sound effects

b) To tell a different story-line.

| *Experiment 25.2 Completed* | ☐ |
|---|---|

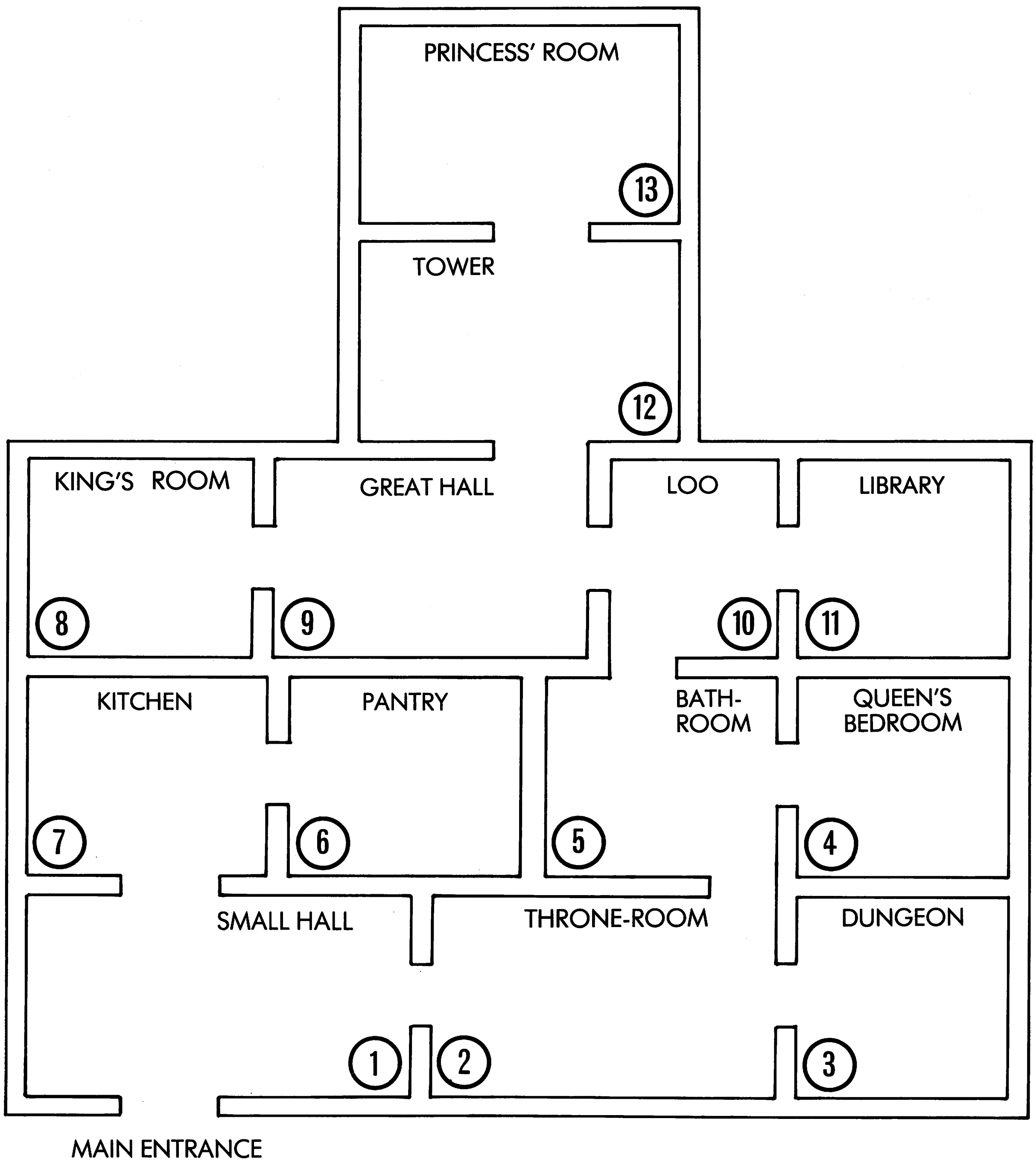
# EXPERIMENT 25·3



The GRAFFS program of the previous experiment offered a simple example of the way in which the structure of a problem can be represented by data. A more complex example is given by the program called DUNGEON, which you may also load and run. This program follows a popular pattern, but it is not as ambitious as some of its commercial equivalents, because it had to be 'shoehorned' into the 35ØØ bytes available on the standard VIC.

When you have played this version of DUNGEON a few times, examine the code carefully and construct your own flow charts. The following information may be useful:

1) Ground plan of the castle:



2) Internal codes:

| | |
|---|---|
| *Rooms* | |
| Outside | Ø |
| Small hall | 1 |
| Throne-room | 2 |
| Dungeon | 3 |
| Queen's bedroom | 4 |
| Bathroom | 5 |
| Pantry | 6 |
| Kitchen | 7 |
| King's room | 8 |
| Great Hall | 9 |
| Loo | 1Ø |
| Library | 11 |
| Tower | 12 |
| Princess's room | 13 |

| | |
|---|---|
| *Perils* | |
| Dragon | 1 |
| Spider | 2 |
| Wasps | 3 |
| Enchantress | 4 |
| Belly-fish | 5 |

| | |
|---|---|
| *Weapons* | |
| Sword | 1 |
| Stave | 2 |
| Fly-spray | 3 |
| Magic Potion | 4 |
| Flamegun | 5 |

Peril number X can only be overcome with weapon X.

## Glossary

| | |
|---|---|
| n: | Number of rooms |
| mm$(5,5): | Scripts for fights with each of the five perils. Thus mm$(1,1) to mm$(1,5) hold the messages for fights with the dragon:<br>"a Dragon"<br>"You fight and"<br>"kill it with your sword"<br>"it kills you"<br>"you both run away!"<br>The battle subroutine (lines 3ØØØ onwards) uses these messages to display accounts of the fights. |
| w$(5): | The *names* of the five weapons. |
| r$(n(: | The *names* of the rooms in the castle. |
| c%(n,4): | The way the rooms are connected. For instance, the row c%(2,1) to c%(2,4) is 5,3,Ø,1. This means that the Throne-room (2) is connected to the bathroom (5) going North, the Dungeon (3) going West, and the Small Hall (3) going East. There is no Southward door in the Throne-room. |
| di$(4): | The names of the four points of the compass: North, East, South and West. |
| d%(5): | The present positions (as room numbers) of the five perils. Thus if d%(2) = 7, this means that the Spider is in the Kitchen! |
| l%(3): | The weapons the hero has with him. |
| w%(5): | The positions of the weapons which the hero has dropped (or hasn't yet picked up). |
| p: | Position of the Princess (as a room number) |
| h: | Position of the Hero (as a room number) |
| q: | 1 if the Princess is with the Hero. Ø if he has not yet found her (or if he has abandoned her). |

Finally, when you feel you really know how the DUNGEON program works, design and write your own program on the same general lines.

| | |
|---|---|
| *Experiment 25.3 Completed* | |

# AFTERWORD

# AFTERWORD



Congratulations! You've reached the end of the course, and if you have followed it carefully and done all the examples, you have been introduced to almost everything there is to be known about the BASIC language. You have begun to appreciate the immense power of the computer to provide delight, pleasure and useful service to all. You have the skill and knowledge to apply your machine in games, in business, in forecasting, in helping the teacher in the classroom, and in many other areas. In all probability, you'd prefer to stop studying and use your expertise in various fascinating and exciting projects you will have dreamed up whilst doing the course.

BASIC is in many respects an excellent language with which to learn programming, but there is one serious drawback which must be mentioned: it is not *standardised*. This means that the version of BASIC you'll find on different machines is generally a little different and usually inferior to VIC BASIC. Some BASICs give you a greatly restricted selection of variable names, and many don't allow you to use arrays of strings. The rules about putting several commands on one line are by no means universal, and the arrangements inside PRINT commands can also vary between machines. There are other slight differences too numerous to mention, and PEEK and POKE have totally different results. If you plan to transfer your programs to any other make of machine, find out its limitations before you design your program, for otherwise you will be in serious difficulty.

You may be glad to hear that this warning doesn't apply to the Commodore PET, which has a form of BASIC almost identical to the VIC. The main differences are that the screen RAM starts in a different place (32768) and the screen itself is a different size. There are no facilities for colour drawings or sound; otherwise programs can be transferred between the two machines. You should also be aware that with expansion memory of more than 3K in the VIC, the screen RAM begins at 4Ø96 and colour at 37888.

We end with ten 'precepts' of good programming. Oscar Wilde once said:

"I always pass good advice on to other people; that's the only thing you can do with it."

It is in this spirit that the following advice is offered; take it or not, as you prefer.

1) Aim for perfection. Every part of your program and its documentation should be as good as you can make it.

2) Design before you build. Decide *what* your program is going to do before thinking about *how* to do it.

3) Be prepared to throw everything away and start again. Don't make the mistake of being in love with what you have already done. Remember that there are lots of ways of solving problems, and the first method you think of is most unlikely to be the best.

4) Plan and record your scheme for allocating variables. The glossary is a working document which should be in constant use when you write your programs.

5) Where it matters, choose good algorithms. For instance use a binary chop in preference to a 'straight through' search, or Quicksort rather than a Bubble sort. It doesn't matter — perhaps because the list to be searched or sorted is very short — always use the *simplest* method you can find.

6) Pay attention to the User Interface. Where it is appropriate, make sure your programs can be used by anyone without special instruction.

7) Use other people's work. Never write a subroutine if you can find a trustworthy one in a program library.

8) Never try anything difficult or complicated, but break the job down into simple steps. Use subroutines and observe the conventions.

9) Avoid "clever programming tricks", especially if you don't understand how they work.

10) Always find another person to do the 'final test' of your program.

Our journey together through BASIC has ended. Good luck and good programming!

Andrew Colin — Glasgow, 1982

# APPENDICES

# APPENDIX A

The VIC computer can be used as a versatile and unusual musical instrument. To get the best performance from the machine you'll need art as well as science. The final test of a music program is not "Is it correct", but "Does it sound good?", and this is ultimately a matter of personal taste.

In the appendix we shall consider

a) Playing a pre-programmed tune

b) Making sounds with different timbres or qualities

c) The production of two-part harmony

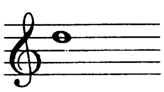
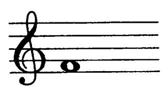
d) Playing directly from the keyboard.

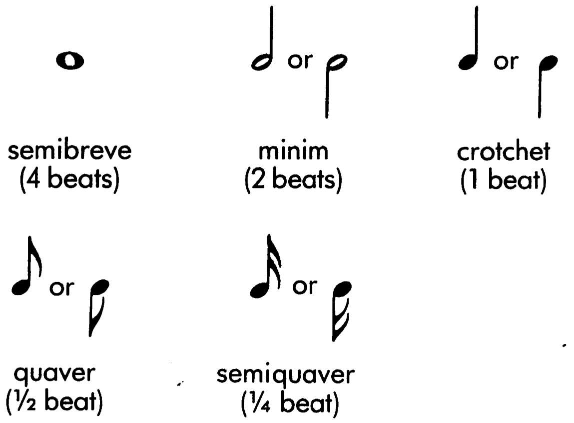
To get some idea of the VIC as an instrument, load and run the program entitled "PRELUDE 1". Try out several different combinations of speed and timbre, and find the one which pleases you most.

To do anything interesting with music, you'll need to know a little about standard musical notation. Many people learn it when they are young, but if need be you'll easily find someone to explain. Anyone who plays a musical instrument can probably help.

A tune is made up of *notes*, generally played one after the other. Every note has four qualities:

a) The *pitch*: this is a measure of whether the note is high or low. In written music, the pitch of a note is shown by its height on the stave, so that

 is higher than 

b) The *duration*: this tells you how long a note is supposed to go on sounding. Written music uses different symbols to show notes of various lengths:




c) The *loudness* of a note also makes a difference to the way it sounds. In musical scores the loudness or volume is indicated by a range of codes:

*ppp pp p mp mf f ff fff*

very soft ◄———————► very loud

d) The *timbre* of a note depends on the instrument on which it is played. Every different instrument has its own special quality which is immediately recognisable. Instruments which are blown or bowed (such as organs, clarinets, horns or 'cellos) produce a *sustained* sound, so that the note is equally loud all the time it is being played. On the other hand instruments which are plucked or hit (harps, pianos or drums) make *decaying* sounds, which start loud and fade away.

On the VIC, sounds are made by POKE'ing numbers into the voice registers at locations 36874 to 36878. The arrangement (as you may remember from book 1) is like this:

| *Register* | *Purpose* |
|---|---|
| 36874 | Bass voice |
| 36875 | Tenor voice |
| 36876 | Treble voice |
| 36877 | Noise generator |
| 36878 | Volume control |

In this section we show how the numbers you POKE and sounds they produce relate to each other.

Every musical sound is made by a *vibrating* object, which could be — for instance — a guitar string, or the diaphragm of a loudspeaker, or the vocal chords in your throat. The vibrations spread through the air as waves, much like ripples on the surface of a pond (but much faster). Eventually the waves reach your ears, where they are heard as a sound.

The pitch of the sound depends only on the *frequency*, or number of vibrations per second. The more vibrations, the higher the note. The pitch is determined by a very simple rule: every doubling of the frequency takes you an octave up and every halving, an octave down.

The frequency of Middle C on the piano is about 264 vibrations per second. The C above Middle C has a frequency of exactly twice 264, or 528 vibrations per second. Similarly, the C *two* octaves above Middle C has a frequency four times 264, or 1056.

The rule also works on the way down; for example, the frequency of the C below Middle C is ½ of 264, or 132.

The octave is divided into much smaller intervals of pitch. Most Western music uses a scale of 12 equal intervals called *semitones*. The frequencies of the notes go up not in equal steps but in a constant *ratio*. The exact value of this ratio is $\sqrt[12]{2}$ (the twelfth root of two) which is about 1.059. The significance of this value is that when you multiply it by itself twelve times you get back to 2.

We can use the value to work out the frequencies of the notes of the scale. To go up a semitone we just multiply the previous frequency by the ratio. If Middle C is 264, then the note one semitone up (C♯ or D♭) will have a frequency of 1.059*264, or (to the nearest whole number) 280. The next note, D, has a frequency of 280*1.059, or 297 (to the nearest whole number).

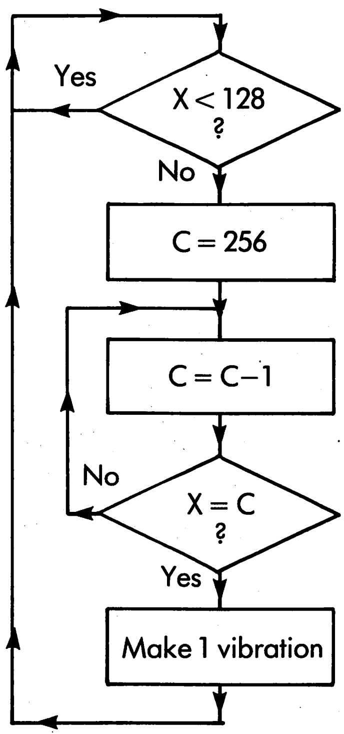
If we apply this rule to all the notes in a scale, we get the following table:

| C | C♯ D♭ | D | D♯ E♭ | E | F | F♯ G♭ | G | G♯ A♭ | A | A♯ B♭ | B | C |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 264 | 280 | 297 | 315 | 334 | 354 | 375 | 397 | 420 | 445 | 471 | 499 | 528 |

The frequency of the high C comes out at 528, just as we expected. Other notes, above the high C or below Middle C, can be worked out in exactly the same way.

Unfortunately you can't tell the VIC voices directly what frequency to play. Each voice works in a way which is best described by a flow diagram:

This loop runs 32768 times per second on the treble voice, 16384 times per second on the tenor voice and 8192 times per second on the bass voice.




**Glossary**

X Number POKE'd into voice register
C Internal variable

The flow chart shows that:

a) If X, the number POKE'd into the voice, is less than 128 then the voice is silent.

b) Otherwise, the voice produces vibrations at a rate which depends on the value of the number POKE'd, and on the speed at which the loop is obeyed. For instance, suppose you POKE 193 into the upper voice. The loop will be obeyed (255−193) or 62 times before a single vibration is produced, and then another 62 times before the next, and so on. Since the 'loop frequency' is 32768, the frequency of the vibrations must be 32768/62, or 528 vibrations per second.

It is important to note that this flow chart doesn't stand for a BASIC program you have to put into the machine; it represents a *mechanism,* which is part of each voice. All three voice mechanisms run at the same time, each one possibly producing a frequency of its own.

A simple bit of algebra will give you a formula to find out what number to POKE for any frequency you may need. It is:

Treble voice: $F = \dfrac{32768}{255-X}$, so $X = 255 - \dfrac{32768}{F}$

Tenor voice: $F = \dfrac{16384}{255-X}$, so $X = 255 - \dfrac{16384}{F}$

Bass voice: $F = \dfrac{8192}{255-X}$, so $X = 255 - \dfrac{8192}{F}$

In each case F is the frequency and X the number you POKE.

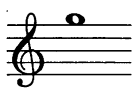
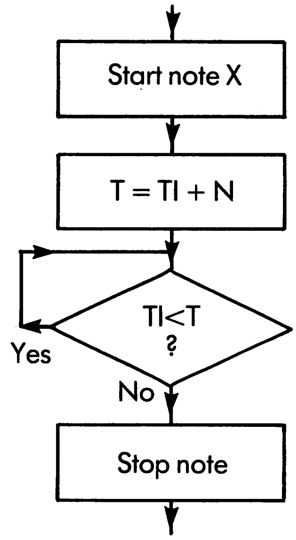
With these formulas, it seems quite straightforward to work out the right value of X for each note in the scale. Unfortunately we soon run into a snag, most of the values turn out not to be whole numbers! For instance, the right frequency for the note

is 792 vibrations per second, and the 'correct' value of X is 213.62. The nearest we can get in practice is 214, and this gives an error of about 1/6 of a semitone — a discrepancy which a trained musician can easily hear.

Figure A1 gives the 'best' X-values for the notes you can play on the treble voice, together with the errors in parts of a semitone. At the lower end the notes are quite well in tune, but as you get towards the higher notes the errors become increasingly obvious. There is no cure, except to keep to the low notes if you find the high ones objectionable.

The notes produced by the other two voices are exactly the same, but displaced one and two octaves down, respectively.

Once a voice has been instructed to sound a note, it keeps playing it until it receives a different command. You can control the duration of a note by a simple loop which uses the internal timer TI. To keep a note going for N jiffies (remember, 60 jiffies make one second) you can use a loop like this:



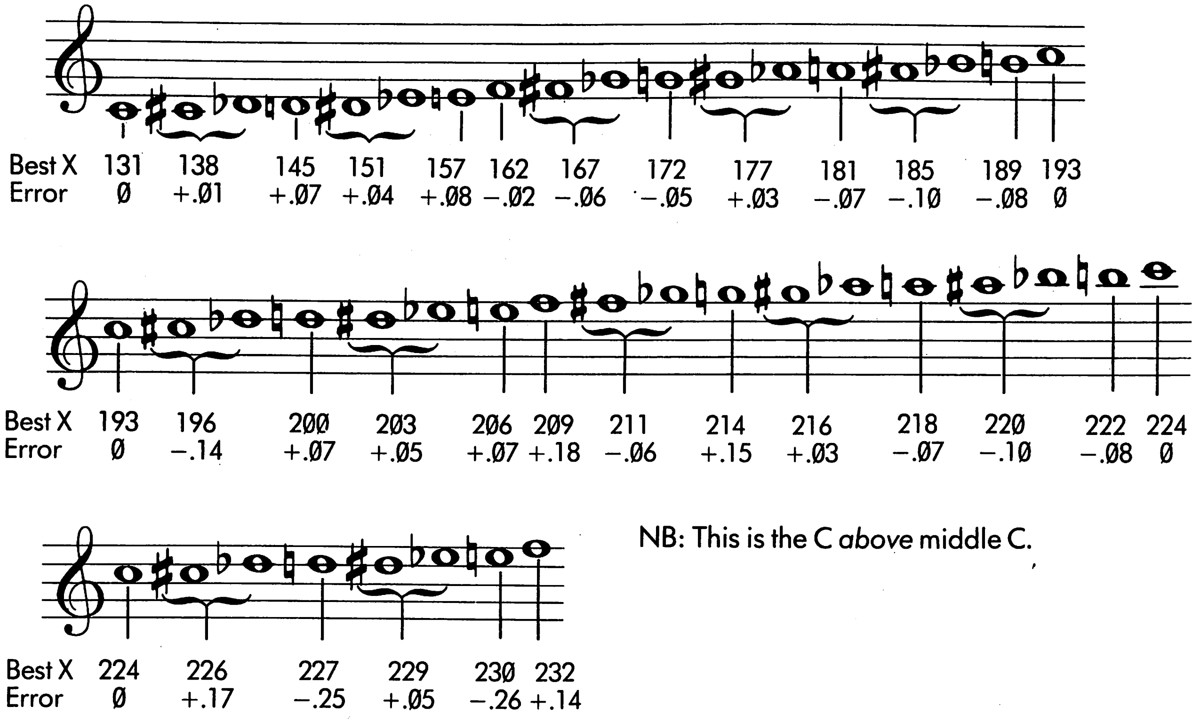

**Glossary**

X: Number POKE'd to voice
N: Duration of note
T: Alarm

or in code:

```
1ØØ POKE 36876,X
11Ø T=TI+N:REM SET ALARM
12Ø IF TI<T THEN 12Ø:REM WAIT
13Ø POKE 36876,Ø:REM STOP NOTE
```

If you forget about loudness and timbre for the moment, you can make the machine play a simple tune just by controlling the pitch and duration of each note. Here is a program to play the first phrase of the tune in Figure A2. The 'beat' is 30 jiffies:




NB: This is the C *above* middle C.

*Figure A1*

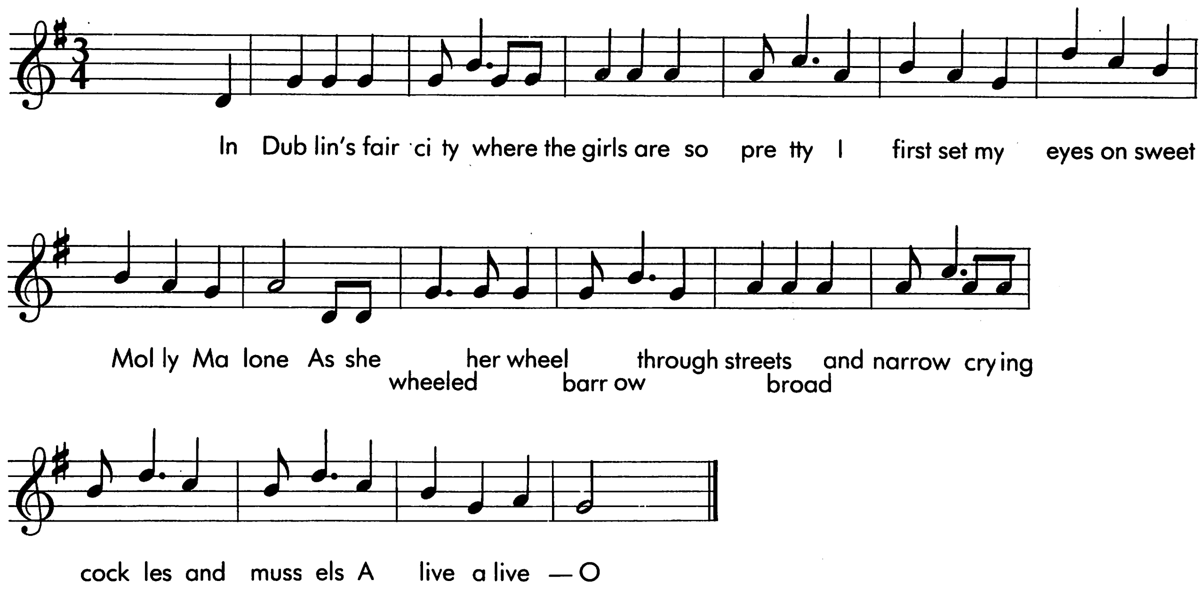



*Figure A2*

```
10 POKE 36878,15 : REM TURN UP
   VOLUME
20 READ X,N
30 IF X = 0 THEN STOP
40 POKE 36876,X : REM START NOTE
50 T = TI + N : REM SET ALARM
60 IF TI<T THEN 60 : REM WAIT FOR
   ALARM
70 POKE 36876,0 : REM STOP NOTE
80 GOTO 20

100 DATA 145,30 : REM IN
110 DATA 172,30 : REM DUB-
120 DATA 172,30 : REM LIN'S
130 DATA 172,30 : REM FAIR
140 DATA 172,15 : REM CI-
150 DATA 189,45 : REM TY
160 DATA 172,15 : REM WHERE
170 DATA 172,15 : REM THE
180 DATA 181,30 : REM GIRLS
190 DATA 181,30 : REM ARE
200 DATA 181,30 : REM SO
210 DATA 181,15 : REM PRET-
220 DATA 193,45 : REM TY

1000 DATA 0,0 : REM END OF TUNE
```

The first DATA statement at line 100 tells the program to put 145 into the treble voice register for 30 jiffies. This plays the first note of the tune. The following notes are played in the same way.

To vary the speed of the music, you can change the number of jiffies used for each note. To avoid changing all the DATA statements you can use a simple 'speed factor'. Alter line 50 to read

```
50 T = B★N + TI
```

and add the line:

```
1 B = 0.5
```

This makes each note last only half the number of jiffies, so the music is played twice as fast. Other values of B will select different speeds again.

# EXPERIMENT A3·1

Extend this program so that it plays the whole of the tune. You will need 36 more DATA statements (or fewer if you have more than 2 numbers per line).

| *Experiment A3.1 Completed* | |
|---|---|

Experiment A3.1 will have convinced you that entering music into the VIC is very hard work and prone to errors. Another drawback is that each DATA statement uses up 16 bytes, and this doesn't leave much room for long tunes.

To make matters a little easier, we'll use a special notation which represents each note as just two characters — one for its pitch and one for its length. The code is set out in Figure A3. It was invented specially for this book and is unlike any other code; in particular, it isn't the same as the code used in the VIC SUPER-EXPANDER. Nevertheless, you will quickly get used to it when you start coding tunes.

As you will see, notes a semitone apart are represented by characters with adjacent ASCII codes. For instance the codes of the symbols ? @ A B C are the numbers 63,64,65,66 and 67. This arrangement makes it easy to look up the right values to be POKEd from a table. A rest (no sound at all) is indicated by the character "(" (left parenthesis).

The lengths of the notes are measured in quarter-notes or semi-quavers. Where a note is more than 9 semiquavers long we simply count on down the ASCII code, so that (for example) a dotted minim 𝅗𝅥. , which is 12 semiquavers long, is shown as "<". Likewise 𝅝 (16 semiquavers) is written as " @ ".

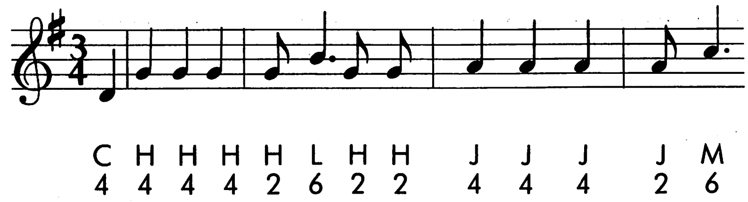
To write down a tune you put down two strings. One string has all the pitch characters, and the second all the length characters. You use as many pairs of strings as you need, and follow the last one with a "Z" by itself.

The coded version of "Dublin's Fair City" is like this:

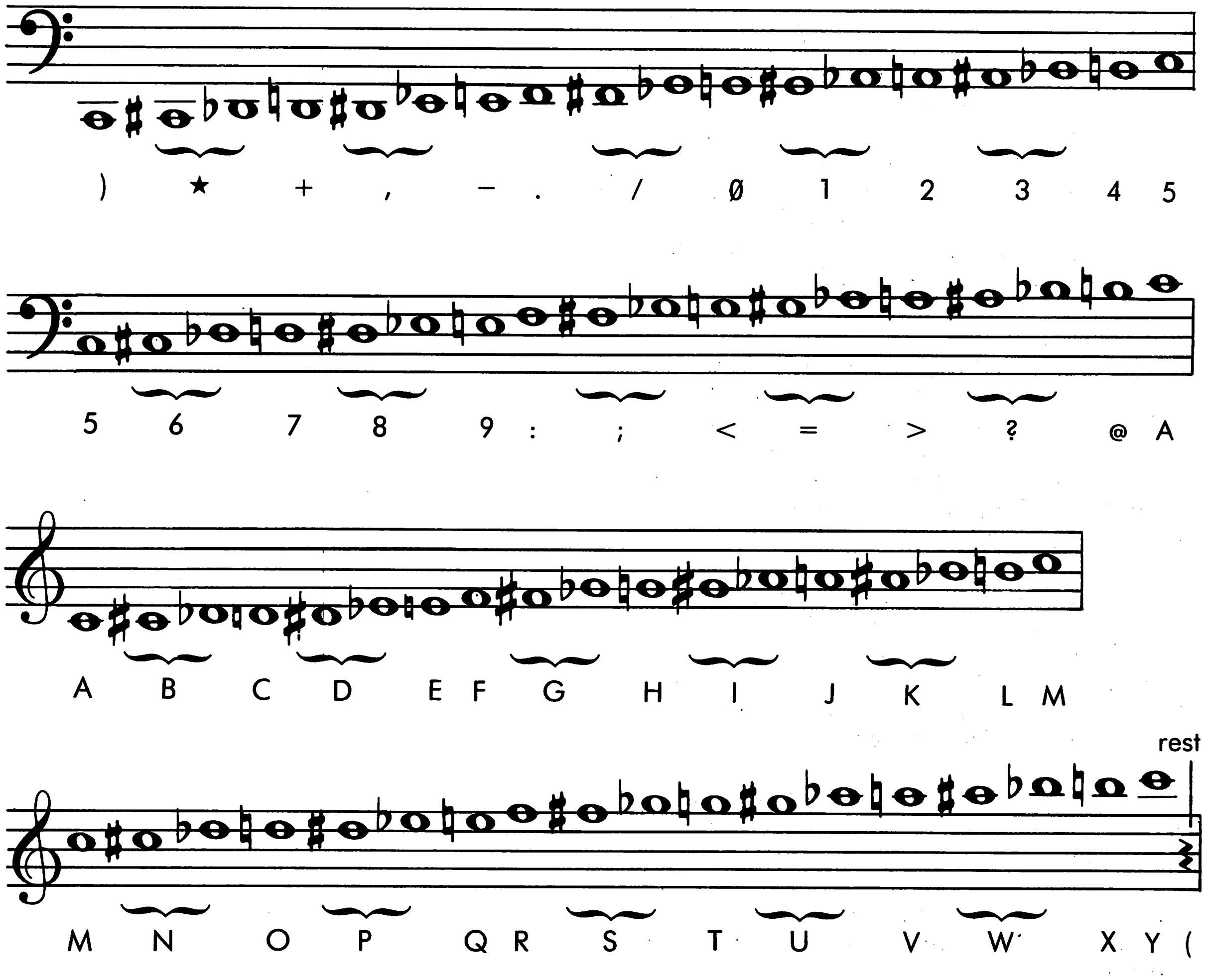
"CHHHHLHHJJJJMJLJHOMLLJHJ"
"444426224442644444444448"

"CCHHHHLHJJJJMJJLOMLOMLHJH"
"2244426444426222642644448"

"Z"



**Pitch:**



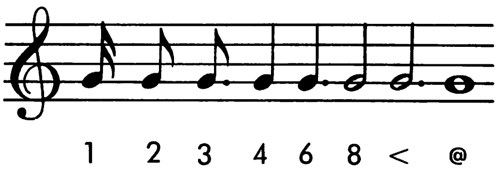
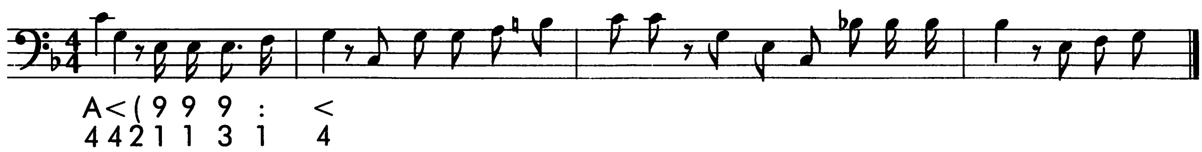
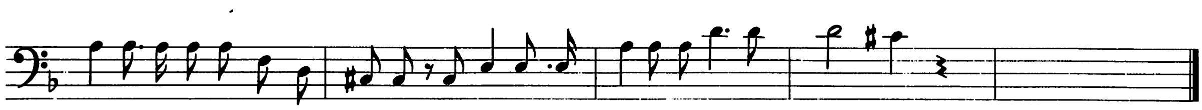

**Duration**


*Figure A3*

---

Another example, which is written in the bass clef and uses some rests, is shown below.

*(from "Elijah" Mendelssohn)*


```
DATA"A<(999:<(5<<> @ AA(<95????(9:<"
DATA"4421131422222222222221146222"

DATA ">>>>>>:766(6999>>>CCCB"
DATA "4312222222224314226284"

DATA "Z"
```

A program which plays tunes written in this code is shown below. The first command at line 1 sets the speed. The DATA lines which follow give the POKE values for all the notes in the scale, and the initial zero is used to produce silence for the rests in the music.

The notation allows a range of four octaves. The program doesn't stick to one voice but chooses the one most appropriate for the pitch of the note being played. This is done in lines 15Ø-18Ø. The numbers 64, 52 and 4Ø are related to the ASCII codes of the characters A, 5, and ).

When music is played with *exactly* the right time values, it often sounds monotonous and boring. You can sometimes make a striking improvement by emphasising the note at the beginning of each bar, by making it sound a little longer than its 'true' value. For instance, you could stretch a crotchet from four semiquavers to five. Don't overdo it, or you will make the music sound ponderous!

Another variation is to gradually slow the music down as you reach the final bars.

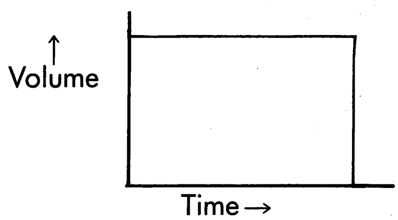
```
  1 R=3
 1Ø DATA Ø,131,138,145,151,157,162,167,
    172,177,181,185,189
 2Ø DATA 193,196,2ØØ,2Ø3,2Ø6,2Ø9,211,
    214,216,218,22Ø,222,224
 3Ø DIM N(25)
 4Ø FOR J = Ø TO 25
 5Ø READ N(J)
 6Ø NEXT J
 7Ø FOR J = 36874 TO 36877 :POKE J,Ø:
    NEXT J :REM SILENCE ALL VOICES
 8Ø POKE 36878,15 : REM TURN UP
    VOLUME CONTROL
 9Ø READ X$
1ØØ IF X$ = "Z" THEN GOTO 2Ø5
11Ø READ Y$
12Ø FOR J = 1 TO LEN(X$) : REM PLAY
    EACH NOTE IN STRING PAIR
13Ø A=ASC(MID$(X$,J,1)) : B= ASC(MID$
    (Y$,J,1))
14Ø T=TI + R★(B−ASC("Ø")) : REM SET
    TIMER
15Ø IF A > = ASC("A") THEN POKE
    36876,N(A−54):GOTO 18Ø
16Ø IF A > = ASC("5") THEN POKE
    36875,N(A−52):GOTO 18Ø
17Ø POKE 36874,N(A−4Ø)
18Ø IF TI < T THEN 18Ø : REM WAIT
19Ø NEXT J
2ØØ GOTO 9Ø
2Ø5 POKE 36878,Ø : STOP : REM TURN
    OFF NOTE
21Ø REM IN DUBLIN'S FAIR CITY
22Ø DATA "CHHHHLHHJJJJMJLJHOMLL
    JHJ","4444262244442644444444448"
23Ø DATA "CCHHHHLHJJJJMJJLOMLO
    MLHJH","2244426444426222642644
    448"
24Ø DATA "Z"
```

You can get an interesting group of effects if you alter the *timbre* or sound quality of the music. So far, we have just turned the sounds on and off, making *sustained* notes. To give the effect of a stringed instrument, you should start the note loudly and taper it off. Key in the following program and try it for yourself:

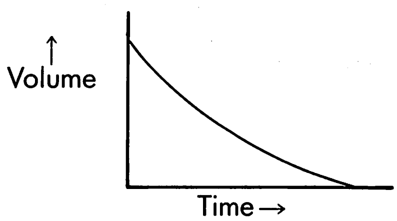


```
 1Ø GET A$
 2Ø IF A$ = "X" THEN P=1 : GOTO 6Ø
 3Ø IF A$ = "Y" THEN P=1.1 : GOTO 6Ø
 4Ø IF A$ = "Z" THEN P= 1.5 : GOTO 6Ø
 5Ø GOTO 1Ø
 6Ø POKE 36876,22Ø : REM START NOTE
 7Ø T = TI + 6Ø : REM SET ALARM ONE
    SECOND LATER
 8Ø J=15 : START LOUDNESS AT 15
 9Ø POKE 36878,J : ADJUST LOUDNESS
1ØØ IF TI>=T THEN 13Ø : REM JUMP IF
   TIME UP
11Ø J = J/P
12Ø GOTO 9Ø
13Ø POKE 36878,Ø : REM STOP NOTE
14Ø GOTO 1Ø
```

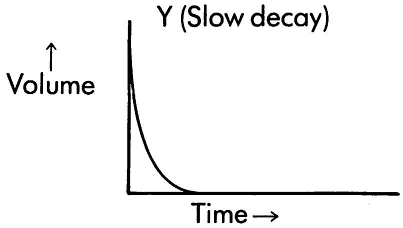
When you start this program nothing happens until you press one of the keys X,Y or Z. X gives a sustained note lasting one second. Y sounds a bit like a harp, and Z produces a muffled note like a mandolin or violin played pizzicato. In all cases, the way the loudness of the note varies with time is called the *envelope*. Graphs of the envelopes used in this program are shown below:



X (Sustained note)



Y (Slow decay)

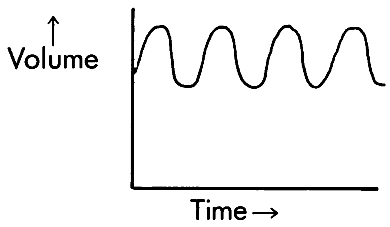


Z (Fast decay)

In the program the change in volume is organised by setting a 'decay factor' in variable P. The variable J, which controls the actual loudness, is adjusted each time round the timing loop.

When P=1, the value of J never varies, so producing a sustained note. When P=.1, the value POKEd into the volume register declines slowly down the sequence of numbers 15,13,12, 11,1Ø,9,8,7,6,6,5,5,4,4,3,3,3,2,2 . . . When P=1.5 the decay is much faster, the numbers being 15,1Ø,6,4,2,1,1,Ø. You can easily experiment with other envelopes either by calculating the loudness values as the note is being sounded or by getting them from a table stored in advance. If you *increase* the volume of each note as it is played, you will get an effect which is impossible to produce on any conventional instrument.

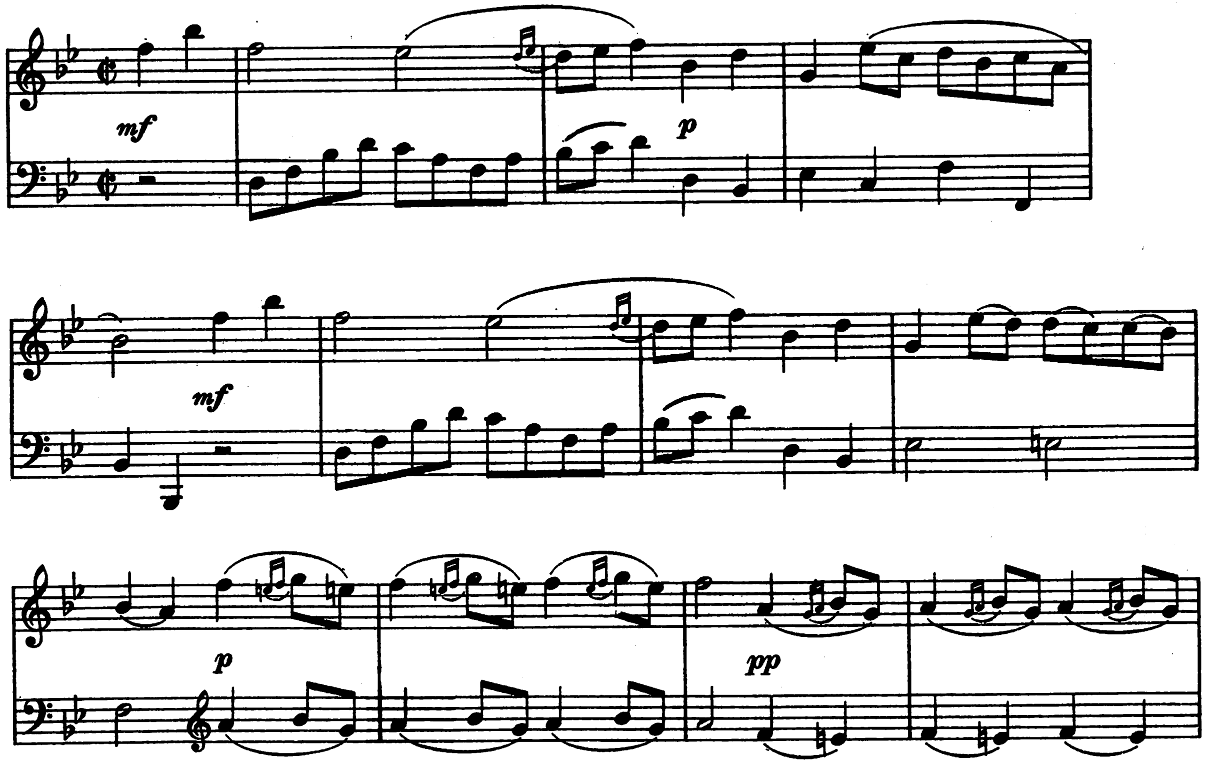
Another type of tonal variation is called 'vibrato'. Its envelope is like this:




To see what it sounds like, load and run the program VIBRATO. You may decide that the use of this timbre is not worth undertaking!

Next we'll consider playing music in two-part harmony. Load and run the program called "GAVOTTE". The piece you hear is by Thomas Arne (1710-1778) and is slightly adapted from the original keyboard version.

Harmonies are played on the VIC by using two or more voices at the same time. The upper part uses the treble voice, and the lower part, either the tenor or bass as most appropriate.

The notation for entering music into the GAVOTTE program is the same as in the previous example. However, each of the two parts uses its own pitch and duration strings, so that at any time the music is the result of information taken from *four* character strings. The first few bars of Arne's Gavotte, and their coding, are


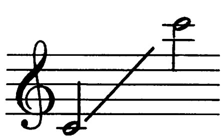

```
1000 DATA "RWRPOPRKOHPMOKMJKRWRPOPRKOHPOOMMKKJ"   } Upper stave
1010 DATA "22441222211111114224411222211111122"   }

1020 DATA "(7:?CA>:>?AC7385: .33(7:?CA>:>?AC7389:"   } Lower stave
1030 DATA "41111111111222222222411111111112224444"   }
```

The final group of strings is followed by four Z's:

```
2000 DATA Z,Z,Z,Z
```

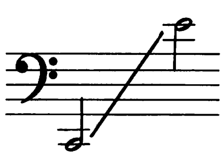
You can erase the DATA statements in the program (from 1000 onwards!) and substitute your own, but there are three special restrictions to watch:

1) The upper voice is limited to the range



C to C

2) The lower voice is limited to the range



C to E

3) The divisions between strings must come in the *same place* for both upper and lower parts.

Unfortunately it is difficult to vary the timbre of music which has two or more parts. This is because the various tone qualities are made by manipulating the volume control register which works on all the voices at the same time. It is impossible, for instance, to play a sustained note in one part whilst a sequence of rapid, decaying notes is played in the other.

Turning to technicalities, the timing of the notes is organised by *two* alarms — one for each part. Once the tune is started, the program waits in a loop for *either* of the alarms to be ready. As soon as one comes up, the program fetches the next note for the corresponding part, starts playing it, and resets the alarm for the end of the new note. The program uses an internal counter instead of the TI timer, because the two parts have an irritating tendency to get slightly out of step with each other. The main control of the program is in lines 9Ø to 19Ø, and the variables have the following uses:

- Z Array of X-values for various notes
- A$ Pitch string for upper stave
- B$ Duration string for upper stave
- C$ Pitch string for lower stave
- D$ Duration string for lower stave
- P Pointer to A$ and B$: Shows which character comes next
- Q Pointer to C$ and D$: Shows which character comes next
- R Alarm for upper stave
- S Alarm for lower stave
- L Internal timer
- V Constant to control rate of playing



# EXPERIMENT A3·2

a) Replace Arne's Gavotte by another piece of your own choice. You might select one of J. S. Bach's two-part Inventions.

b) Modify the program so that you can control the loudness of the music being played.
*Hint:*
*You will probably need a fifth, 'volume-control' string.*

*Experiment A3.2 Completed*

Finally, we'll look at a program which allows the VIC to be used as a keyboard instrument. Load and try out the "KEYBOARD" program. You will soon discover that the VIC keyboard has been made to look as much like a piano as possible. The whole of the second row of keys is used for the 'white' notes, and some of the keys in the upper row stand for the 'black' ones.

The VIC instrument is fitted with two more controls:

a) The Volume can be turned up by repeatedly pressing A, and turned down by pressing S. The maximum volume is 15, and the minimum, zero.

b) The decay factor for the notes can be decreased by typing K, and increased by typing L. This allows the tone quality to vary from a broad sustained tone to a muffled 'plink'.

If you list and examine the program, you will see that the tables are somewhat irregular because the notes are assigned to the characters almost at random from the computer's point of view. Otherwise the program is straightforward, and easily modified to give, say — an optional vibrato. Another modification is to make it remember the notes of the tune you play by storing the X-values and timings in an array. The machine can then repeat your performance later.

This appendix has given you only the briefest of introductions to the musical potential of the VIC. The machine can be extremely rewarding in composition as well as performance, and the only important limits are set by your skill and artistry as a programmer and musician.

# APPENDIX B



## PROGRAM LIBRARY

This section contains a small library of useful subroutines. All the subroutines have been carefully checked. As a rule they do not alter the values of their input parameters, unless the same variables are also specified as output parameters. (One exception is the subroutine for solving simultaneous equations.)

When you are designing a new program you should select and incorporate any of these subroutines you may need; your program will be more reliable and be working sooner as a result.

There are two ways of using the subroutines:

a) If you have a minimal VIC (with 3.5K of store), copy the subroutines from the text which follows. You are strongly advised to check the accuracy of your work by keying in the 'driver' program given with each subroutine and checking that you get the same (or similar) results. The driver program can be erased once you are satisfied.

b) If you have a 3K extra store module (or more) you can use the LIBRARY program. Load it from the cassette tape, and run through the menu answering 'YES' or 'NO' to each subroutine. After the last item the LIBRARY program will erase itself and leave only the subroutines you actually need. At this point you must SAVE them on to a separate tape.

   *Note that the LIBRARY program cannot be restarted except by reloading from tape. If interrupted in mid-stream, it may leave the VIC in a non-standard state, unable to load programs correctly. This can always be cured by switching the computer off and on again.*

The subroutines have been arranged in 4 sections:

Section A: Keyboard Input.

1. Tolerant input: Accepts I and O instead of 1 and Ø, and gives clear error messages.
2. Robust input: Does not respond to any except legitimate characters.

Section B: Screen Output.

3. Bigletters: Displays text four times usual size.
4. Formatted output: Displays numbers with user-specified number of decimal places.
5. String display: Displays a long string without breaking words over lines.
6. Binary converter: Displays number as a binary pattern.

Section C: Internal Manipulation.

7. Extract surname: Extracts surname from a string holding a person's full name.
8. List search: Searches a list for a specific entry.
9. Bubble sort: Sorts a small number of strings into alphabetical order.
10. Quick sort: Sorts a list of numbers (or strings) into order.

Section D: Mathematics.

11. Fraction simplifier: Reduces fraction to its lowest terms.
12. Simultaneous equations: Solves simultaneous equations.

### 1. TOLERANT INPUT

*Purpose:* To input numbers from an unskilled user. All spaces are ignored, and letters I and O are taken as digits one and zero. All other errors are clearly explained.

*Lines: 45ØØ-466Ø.*

*Parameters: Output:* Result is delivered in X1.

*Local Variables:* XX$, YY$, JJ, CC$.

```
45ØØ REM TOLERANT INPUT OF NUMBERS
451Ø INPUT XX$
452Ø YY$ = ""
453Ø FOR JJ = 1 TO LEN (XX$)
454Ø CC$ = MID$ (XX$,JJ,1)
455Ø IF CC$ = "O" THEN YY$ = YY$ + "Ø":
       GOTO 46ØØ: REM REPLACE LETTER O
       BY DIGIT Ø
456Ø IF CC$ = "I" THEN YY$ = YY$ + "1":
       GOTO 46ØØ
457Ø IF CC$ = " " THEN 46ØØ
458Ø IF NOT (CC$ <= "9" AND CC$ >=
       "Ø" OR CC$ = "+" OR CC$ = "-"
       OR CC$ = ".") THEN 462Ø
459Ø YY$ = YY$ + CC$
46ØØ NEXT JJ
461Ø X1 = VAL (YY$): RETURN
462Ø PRINT "NUMBERS CONSIST OF"
```

```
4630 PRINT "DECIMAL DIGITS Ø-9,"
4640 PRINT "+,– AND . ONLY"
4650 PRINT "PLEASE TRY AGAIN"
4660 GOTO 451Ø
```

**Driver program:**

```
1Ø GOSUB 45ØØ
2Ø PRINT "VALUE ="; X1
3Ø GOTO 1Ø
```

Test results:

```
RUN
? 778
VALUE = 778
? IOI
VALUE = 1Ø1
? –34·56
VALUE = –34·56
? 45.K
NUMBERS CONSIST OF
DECIMAL DIGITS Ø-9,
+,– AND · ONLY
PLEASE TRY AGAIN
? 7·7
VALUE = 7·7
```

## 2. ROBUST INPUT

*Purpose:* To read a number from the keyboard, ignoring all meaningless characters. DEL may be used and number is ended by RETURN.

*Lines:* 7ØØØ-7Ø9Ø

*Parameters: Output:* Result delivered in X1.

*Local Variables:* PP, AA$, XX$.

```
7ØØØ REM ROBUST NUMBER INPUT
7Ø1Ø XX$ = " ":PP=Ø
7Ø2Ø GET AA$: IF AA$ = " " THEN 7Ø2Ø
7Ø3Ø IF AA$>= "Ø" AND AA$ <= "9"
     THEN PRINT AA$;: XX$ = XX$ + AA$:
     PP = PP+1: GOTO 7Ø2Ø
7Ø4Ø IF ASC(AA$) <> 2Ø THEN 7Ø7Ø:
     REM LOOK FOR DEL
7Ø5Ø IF PP=Ø THEN 7Ø2Ø: REM CAN'T
     ERASE NOTHING
7Ø6Ø PRINT " [SHIFT] and [CRSR ⇐⇒] space
     [SHIFT] and [CRSR ⇐⇒] ";: PP=PP–1:
     XX$=LEFT$(XX$,PP): GOTO 7Ø2Ø
7Ø7Ø IF ASC (AA$) <> 13 THEN 7Ø2Ø: REM
     LOOK FOR RETURN
7Ø8Ø IF PP=Ø THEN 7Ø2Ø:REM MUST BE
     SOME DIGITS
7Ø9Ø X1=VAL (XX$): RETURN
```

**Driver program:**

```
1Ø GOSUB 7ØØØ
2Ø PRINT X1
3Ø GOTO 1Ø
```

## 3. BIG LETTERS

*Purpose:* To Display VIC characters four times their usual size.

*Lines:* 8ØØØ to 82ØØ.

*Parameters: Input:* The next character to be displayed is supplied in A1$. It may be any printable character or space, RETURN, CLR/HOME, a colour code, CTRL and RVS ON or CTRL and RVS OFF.

*Local Variables:* AA, BB, JJ, KK, LL, MM, NN, QQ.



*Note: QQ keeps track of the current RVS status and must not be used outside the subroutine.*

```
8ØØØ REM DISPLAY CHARACTER IN A1$
     FOUR TIMES USUAL SIZE
8Ø1Ø BB = ASC (A1$)
8Ø2Ø IF BB = 13 OR BB = 141 THEN PRINT
     " [CRSR ⇑⇓] [CRSR ⇑⇓] [CRSR ⇑⇓] ": RETURN
8Ø3Ø IF BB = 18 THEN QQ = 1: RETURN
8Ø4Ø IF BB = 146 THEN QQ = Ø: RETURN
8Ø5Ø IFBB<32THENPRINTMID$(" [CTRL] and
[RVS OFF] 5 times [CTRL] and [" 2] [CTRL]
and [RVS OFF] 13 times [CLR HOME] [CTRL] and
[RVS OFF] 8 times [CTRL] and [" 2] [CTRL]
and [RVS OFF] [CTRL] and [& 6] [CTRL]
and [' 7] ",BB+1,1);:RETURN
8Ø6Ø IFBB>=144 AND BB<16ØTHENPRINT
     MID$(" [CTRL] and [! 1] [CTRL]
and [RVS OFF] [CTRL] and [RVS OFF] [SHIFT]
and [CLR HOME] [CTRL] and [RVS OFF] 3 times
[SPACE] [SPACE] [SPACE] [SPACE]
[CTRL] and [% 5] [CTRL] and [RVS OFF]
[CTRL] and [( 8] [CTRL] and [$ 4]
",BB–143,1);: RETURN
8Ø7Ø AA= (BB AND 31)+Ø·5 ★ (BB AND
       128): IF (BB AND 64)=Ø THEN
     AA=AA+32
8Ø8Ø FOR JJ=Ø TO 6 STEP 2
8Ø9Ø KK=PEEK (32768+8★AA+JJ):
     LL=PEEK (32769+8★AA+JJ)
81ØØ NN=64: FOR MM=Ø TO 3
811Ø PP = 1+8★ INT (KK/NN)+2★INT
     (LL/NN)
812Ø KK=KK–INT (KK/NN)★NN:LL=LL–
     INT (LL/NN)★NN
```

```
8130 IFQQ=ØTHENPRINTMID$(" [CTRL]
and [RVS OFF] [SPACE] [CTRL] and [RVS OFF]
[C=] and [D] [CTRL] and [RVS OFF] [C=]
and [F] [CTRL] and [RVS OFF] [C=] and
[I] [CTRL] and [RVS OFF] [C=] and [C]
[CTRL] and [RVS ON] [C=] and [K]
[CTRL] and [RVS ON] [C=] and [B]
[CTRL] and [RVS ON] [C=] and [V]
[CTRL] and [RVS OFF] [C=] and [V]
[CTRL] and [RVS OFF] [C=] and [B]
[CTRL] and [RVS OFF] [C=] and [K]
[CTRL] and [RVS ON] [C=] and [C]
[CTRL] and [RVS ON] [C=] and [I]
[CTRL] and [RVS ON] [C=] and [F]
[CTRL] and [RVS ON] [C=] and [D]
[CTRL] and [RVS ON] [SPACE]

",PP,2);:GOTO815Ø

814Ø PRINTMID$(" [CTRL] and [RVS ON]
[SPACE] [CTRL] and [RVS ON] [C=] and [D]
[CTRL] and [RVS ON] [C=] and [F]
[CTRL] and [RVS ON] [C=] and [I]
[CTRL] and [RVS ON] [C=] and [C]
[CTRL] and [RVS OFF] [C=] and [K]
[CTRL] and [RVS OFF] [C=] and [B]
[CTRL] and [RVS OFF] [C=] and [V]
[CTRL] and [RVS ON] [C=] and [V]
[CTRL] and [RVS ON] [C=] and [B]
[CTRL] and [RVS ON] [C=] and [K]
[CTRL] and [RVS OFF] [C=] and [C]
[CTRL] and [RVS OFF] [C=] and [I]
[CTRL] and [RVS OFF] [C=] and [F]
[CTRL] and [RVS OFF] [C=] and [D]
[CTRL] and [RVS OFF] [SPACE] ",PP,2);
815Ø NN=INT(NN/4):NEXT MM
816Ø PRINT" [CRSR ↑↓] [SHIFT] and [CRSR ⇐⇒] 4 times";
817Ø NEXTJJ
818Ø PRINT" [SHIFT] and [CRSR ↑↓] 4 times [CRSR ⇐⇒]
    4 times ";
819Ø IFPEEK(211)>18THENPRINT" [CRSR ↑↓]
    3 times "
82ØØ RETURN
```

**Driver program**

```
1Ø GET A1$: IF A1$= " " THEN 1Ø
2Ø GOSUB 8ØØØ
3Ø GOTO 1Ø
```

*Sample run:* Not reproducable. (Try it and see.)

## 4. FORMATTED NUMBER

*Purpose:* To display a number in a controlled format.

*Lines:* 5ØØØ to 513Ø.

*Parameters: Input:* Number to be displayed in X1
Number of decimal places required in Y1.

*Local Variables:* NN$, PP, JJ, XX.

*Note: If the number to be displayed is greater than 999999999 then it is displayed in floating format without rounding. Otherwise it is rounded and displayed with Y1 places after the decimal point. If Y1 = Ø the number is rounded to the nearest integer.*

```
5ØØØ REM DISPLAY X1 TO Y1 DECIMAL
       PLACES
5Ø1Ø XX=X1: IF Y1 >Ø AND ABS(XX) <=
       999999999 THEN 5Ø5Ø
5Ø2Ø XX=XX+0·5
5Ø3Ø PRINT INT (XX);
5Ø4Ø RETURN
5Ø5Ø IF XX <Ø THEN XX=XX−Ø·5★1Ø↑−
       Y1:GOTO 5Ø7Ø
5Ø6Ø XX=XX+Ø·5★1Ø↑−Y1
5Ø7Ø NN$= STR$ (XX)
5Ø8Ø FOR PP = 1 TO LEN (NN$)
5Ø9Ø IF MID$(NN$, PP, 1) = "." THEN
       PRINT LEFT$(NN$, PP+Y1);: RETURN
51ØØ NEXT PP
511Ø PRINT NN$; ".";
512Ø FOR JJ=1 TO Y1: PRINT "Ø";: NEXT JJ
513Ø RETURN
```

**Driver Program:**

```
1Ø FOR J= 667 TO 67Ø
2Ø FOR Y1=Ø TO 3
3Ø X1= SQR(J)
4Ø GOSUB 5ØØØ
5Ø NEXT Y1
6Ø PRINT
7Ø NEXT J
8Ø STOP
```

*Test Run:*

| | | | |
|---|---|---|---|
| 26 | 25·8 | 25·83 | 25·826 |
| 26 | 25·8 | 25·85 | 25·846 |
| 26 | 25·9 | 25·87 | 25·865 |
| 26 | 25·9 | 25·88 | 25·884 |

## 5. STRING DISPLAY

*Purpose:* To display a string without splitting words over lines.

*Lines:* 57ØØ-58ØØ.

*Parameters: Input:* String to be displayed in X1$.

*Local Variables:* XX$, PP, QQ, RR.

```
57ØØ REM DISPLAY X1$ WITHOUT
     SPLITTING WORDS
571Ø XX$=X1$
572Ø PP=LEN(XX$)
573Ø IF PP <=22 THEN RR=PP:GOSUB
     578Ø: RETURN
574Ø FOR QQ=23 TO 1 STEP -1
575Ø IF MID$(XX$,QQ,1) = " " THEN
     RR=QQ-1:GOSUB 578Ø: XX$=
     RIGHT$(XX$, PP-QQ):GOTO
     572Ø
576Ø NEXT QQ
577Ø RR=22: GOSUB 578Ø: XX$=RIGHT$
     (XX$, PP-22): GOTO 572Ø
578Ø REM INTERNAL SUBROUTINE
579Ø PRINT LEFT$(XX$, RR);: IF RR<22
     THEN PRINT
58ØØ RETURN
```

**Driver program:**

```
1Ø X1$= "TYPE ANY STRING
   AT ALL UP TO THREE LI
   NES LONG TO TRY OUT TH
   E SUBROUTINE"
2Ø GOSUB 57ØØ
3Ø INPUT X1$: GOSUB 57ØØ
4Ø GOTO 3Ø
```

*Sample output:* Try it and see!

## 6. BINARY CONVERTER

*Purpose:* To display the binary pattern of a number in the range Ø-255.

*Lines:* 1ØØØ-1Ø6Ø.

*Parameters: Input:* Number to be displayed in X1.

*Local Variables:* YY, KK, XX.

```
1ØØØ REM CONVERT X1 TO BINARY AND
     DISPLAY
1Ø1Ø YY=256: XX=X1: FOR KK=1 TO 8
1Ø2Ø YY=YY/2
1Ø3Ø IF XX>=YY THEN XX=XX-YY:
     PRINT "★";:GOTO 1Ø5Ø
1Ø4Ø PRINT " ";
1Ø5Ø NEXT KK
1Ø6Ø PRINT: RETURN
```

**Driver program:**

```
1Ø FOR J=Ø TO 9
2Ø X1=J: GOSUB 1ØØØ
3Ø NEXT J
4Ø STOP
```

*Sample output:*

```

           ★
        ★
        ★  ★
     ★
     ★     ★
     ★  ★
     ★  ★  ★
  ★
  ★        ★
```

## 7. EXTRACT SURNAME

*Purpose:* To extract a surname from a person's full name.

*Lines:* 41ØØ-42ØØ.

*Parameters: Input:* A person's name, in NI$, in any of the following forms:
J. X. SMITH
GEORGE ELLIOTT
ALVA T EDISON
WELLINGTON-COO
K. O'SHAUGHNESSY

*Output:* The person's surname, in YI$. The surname is defined as the unbroken sequence of letters, hyphens and apostrophes nearest the end of the string in NI$.

Examples are: SMITH
ELLIOTT
EDISON
WELLINGTON-COO
O'SHAUGNESSY

If a surname can't be found, YI$ is delivered empty.



*Local Variables:* JJ, KK, CC$.

*Note: This subroutine works correctly for European names, but would require modification for names from other parts of the world — eg. China.*

```
4100 REM EXTRACT SURNAME FROM NI$
     AND DELIVER IN YI$
4110 JJ=LEN (NI$)
4120 IF JJ=0 THEN YI$= "": RETURN
4130 IF MID$(NI$, JJ, 1) <"A" OR MID$
     (NI$, JJ, 1) > "Z" THEN JJ=JJ-1:
     GOTO 4120
4140 FOR KK=JJ TO 1 STEP -1
4150 CC$=MID$(NI$, KK, 1)
4160 IF NOT (CC$ >= "A" AND CC$ <=
     "Z" OR CC$= "-" OR CC$= "'")
     THEN 4190
4170 NEXT KK
4180 KK=0
4190 YI$=MID$(NI$, KK+1, JJ -KK)
4200 RETURN
```

**Driver program:**

```
10 INPUT "NAME PLEASE"; NI$
20 GOSUB 4100
30 PRINT "SURNAME IS  "; YI$
40 GOTO 10
```

*Test run:* NAME PLEASE? J. X. SMITH
SURNAME IS SMITH
NAME PLEASE? GEORGE ELLIOTT.
SURNAME IS ELLIOTT
NAME PLEASE? WELLINGTON-COO
SURNAME IS WELLINGTON-COO
NAME PLEASE? K. O'SHAUGHNESSY
SURNAME IS O'SHAUGHNESSY

## 8. LIST SEARCH

*Purpose:* To search an ordered list for a specific entry, using the binary chop method.

*Lines:* 6000-6050.

*Parameters: Input:* List to be searched (must be in increasing alphabetical order) in AI$
Index of first entry in L1; index of last entry in H1
String to be looked for in XI$.

*Output:* If the entry is found, M1 holds its index; if not found M1=-1.

*Local Variables:* HH, LL.

```
6000 REM SEARCH ORDERED LIST AI$
6005 HH=H1: LL=L1
6010 IF HH<LL THEN M1=-1: RETURN
6020 M1=INT (0·5*(HH+LL))
6030 IF XI$= AI$(M1) THEN RETURN
6040 IF XI$ < AI$(M1) THEN HH=M1-1:
     GOTO 6010
6050 LL=M1+1: GOTO 6010
```

**Driver program:**

```
10 DATA ABLE, BAKER, CHARLIE, DOG, ERNIE,
   FRED, GORDON
20 DATA HARRY, IONA, JILL, KATE, LYDIA,
   MURIEL, NICK
30 DIM AI$(14)
40 FOR J=1 TO 14: READ AI$(J): NEXT J
50 INPUT "TYPE A NAME";XI$
60 H1=14: L1=1: GOSUB 6000
70 IF M1<0 THEN PRINT "NOT FOUND":
   GOTO 50
80 PRINT "FOUND AT ENTRY"; M1: GOTO 50
```

*Sample Run:* TYPE A NAME? CHARLIE
FOUND AT ENTRY 3
TYPE A NAME? DAVID
NOT FOUND
TYPE A NAME? NICK
FOUND AT ENTRY 14
....

## 9. BUBBLE SORT

*Purpose:* To sort a few string items into alphabetical order.

*Line Numbers:* 6500-6560.

*Parameters: Input:* List of items to be sorted in AI$(1) to AI$(N1).
Number of items in N1.

*Output:* Sorted list in AI$ (1) to AI$ (N1).

*Local Variables:* KK, DD$, SS$.

```
6500 REM BUBBLE SORT OF STRINGS IN
     AI$
6510 SS$= "NO"
6520 FOR KK=1 TO N1-1
```

```
6530 IF AI$(KK)>AI$(KK+1) THEN DD$=
     AI$(KK): AI$(KK)=AI$(KK+1):
     AI$(KK+1)=DD$: SS$= "YES"
6540 NEXT KK
6550 IF SS$= "YES" THEN 6510
6560 RETURN
```

**Driver program:**

```
10 INPUT "N1"; N1
20 PRINT "TYPE "; N1; "WORDS"
30 DIM AI$ (N1)
40 FOR J=1 TO N1: INPUT AI$(J): NEXT J
50 GOSUB 6500
60 FOR J=1 TO N1: PRINT AI$(J): NEXT J
70 STOP
```

*Sample Run:*
```
RUN
N1 ?7
TYPE 7 WORDS
? PEARS
? CHERRIES
? BANANAS
? ORANGES
? DATES
? PLUMS
? APPLES
APPLES
BANANAS
CHERRIES
DATES
ORANGES
PEARS
PLUMS
```

## 10. QUICKSORT

*Purpose:* To sort items into numerical order, using Hoare's Quicksort algorithm.

*Line Numbers:* 6200-6380.

*Parameters: Input:* List of numbers to be sorted in A1(1) to A1(N1).
*Output:* Sorted list appears in A1(1) to A1(N1).

*Local Variables:* SS, SS%, AA, BB, XX, YY, ZZ, DD, PP.

*Notes:* (i) *SS must not be used anywhere else in the program if the sort subroutine is used more than once.*
(ii) *The subroutine will sort strings instead of numbers if the following substitutions are made throughout:*
*AI$ for A1: ZZ$ for ZZ; DD$ for DD*
(iii) *If the routine is being used to sort a set of records with the fields spanning several arrays then the following statements should be altered to ensure that all the fields of every record are moved:*
*6280 6290*
(iv) The comparison operation in statements 6260 and 6270 may be inverted or replaced if a different ordering is required.

```
6200 REM QUICKSORT
6210 IF SS=1 THEN 6230
6220 DIM SS%(100):SS=1: REM DECLARE
     STACK
6230 AA=1: BB=N1: SS%(0)=1: PP=1
6240 XX=AA: YY=BB: ZZ=A1(BB)
6250 IF XX >= YY THEN 6290
6260 IF A1(XX) <= ZZ THEN XX=XX+1:
     GOTO 6250
6270 IF A1(YY) >= ZZ THEN YY=YY-1:
     GOTO 6250
6280 DD=A1(YY): A1(YY)=A1(XX): A1(XX)=
     DD: GOTO 6250
6290 A1(BB)=A1(XX): A1(XX)=ZZ
6300 IF XX-AA <=1 THEN 6340
6310 SS%(PP)=XX: SS%(PP+1)=BB:
     SS%(PP+2)=2: PP=PP+3
6320 BB=XX-1: GOTO 6240
6330 PP=PP-3: XX=SS%(PP): BB=SS%
     (PP+1)
6340 IF BB-XX <=1 THEN 6370
6350 SS%(PP)=3: PP=PP+1: GOTO 6240
6360 PP=PP-1
6370 ON SS%(PP-1) GOTO 6380, 6330,
     6360
6380 RETURN
```



**Driver program:**

```
10 INPUT "HOW MANY";N1
20 DIM A1(N1)
30 FOR J=1 TO N1: A1(J)=INT(1000*RND(0)):
   NEXT J
40 GOSUB 6200
50 FOR J=1 TO N1: PRINT A1(J);: NEXT J
60 STOP
```

*Sample run:*
```
HOW MANY? 6
331   342   369   540   870   912
```
(ie. 6 numbers in ascending order).

## 11. FRACTION SIMPLIFIER

*Purpose:* To reduce fractions to their lowest terms.

*Line Numbers:* 5500-5630.

*Parameters: Input:* A1 (Top of fraction)
B1 (Bottom of fraction)
*Output:* C1 (Top of simplified fraction)
D1 (Bottom of simplified fraction)

*Local Variables:* JJ, KK.

```
5500 REM REDUCE FRACTION A1/B1 TO
     ITS LOWEST TERMS
```

```
551Ø REM RESULT IN C1/D1. LOCALS ARE
     JJ, KK
552Ø REM ERROR IF AI OR B1 NOT
     WHOLE NUMBERS OR IF B1 < 1
553Ø IF AI = INT(AI) AND B1 = INT(B1)
     AND B1 > 1 THEN 555Ø
554Ø PRINT "WRONG PARAMETERS TO
     FRACTION SIMPLIFIER"; AI;B1: STOP
555Ø IF AI=Ø THEN C1=Ø: D1=1: RETURN
556Ø JJ=AI: KK=B1
557Ø IF AI < Ø THEN JJ = −AI
558Ø IF KK=Ø THEN 562Ø
559Ø IF JJ=Ø THEN JJ=KK: GOTO 562Ø
56ØØ IF JJ>KK THEN JJ=JJ−INT(JJ/KK)★
     KK: GOTO 558Ø
561Ø KK=KK−INT (KK/JJ)★JJ: GOTO
     558Ø
562Ø C1=A1/JJ: D1=B1/JJ
563Ø RETURN
```

**Driver program:**

```
1Ø INPUT "GIVE FRACTION"; A1,B1
2Ø GOSUB 55ØØ
3Ø PRINT C1;"/"; D1
4Ø GOTO 1Ø
```

*Sample run:*

```
RUN
GIVE FRACTION? 2, 4
1/2
GIVE FRACTION? 123, 456
41/152
GIVE FRACTION? 375, 1ØØØ
3/8
GIVE FRACTION? Ø, 1234
Ø/1
GIVE FRACTION? Ø·5, Ø·75
WRONG PARAMETERS TO FR
ACTION SIMPLIFIER ·5
·75
RUN
GIVE FRACTION? −50, 100
−1/2
GIVE FRACTION? 77, Ø
WRONG PARAMETERS TO FR
ACTION SIMPLIFIER 77
Ø
```

## 12. SIMULTANEOUS EQUATIONS

*Purpose:* To solve simultaneous equations, N1 eg. equations in N1 unknowns.

*Lines:* 9ØØØ-927Ø.

*Parameters:* *Input:* N1: The number of equations A1(1,1) to A1 (N1,N1); A two-dimensional array which holds the matrix of co-efficients B1(1) to B1 (N1): The vector of right-hand sides.
*Output:* X1(1) to X1(N1) holds the vector of solutions.

*Local Variables:* DD, JJ, KK, LL.

*Note: The initial values in the arrays A1 and B1 are destroyed.*

*Example:* Consider three equations in three unknowns:

$$3x + 2y + 1z = 19$$
$$2x + 7y + 2z = 55$$
$$4x + 1y + 4z = 19$$

A program to solve these equations could read as follows:

```
1Ø DATA 3, 2, 1, 19
2Ø DATA 2, 7, 2, 55
3Ø DATA 4, 1, 4, 19
4Ø DIM A1(3,3), B1(3), X1(3)
5Ø N1=3
6Ø FOR J=1 TO N1: FOR K=1 TO N1:
   READ A1(K,J): NEXT K: READ B1(J):
   NEXT J
7Ø GOSUB 9ØØØ: REM CALL
   SUBROUTINE
8Ø FOR J=1 TO N1: PRINT X1(J):
   NEXT J
9Ø STOP
```

*Timing:* The time needed to solve a set of N1 equations is roughly proportional to the cube of N1. Typical figures are:

| N1 | Time (seconds) |
|---|---|
| 5 | 2 |
| 1Ø | 1Ø |
| 15 | 3Ø |
| 2Ø | 65 |
| 25 | 121 |

```
9ØØØ REM SOLVE N1 SIMULTANEOUS
     EQUATIONS A1.X1=B1
9Ø1Ø IF N1=1 THEN X1(1)=B1(1)/A1(1,1):
     RETURN
9Ø2Ø FOR JJ=1 TO N1−1: REM FIND
     PIVOT
9Ø3Ø DD=ABS(A1(JJ,JJ)): LL=JJ
9Ø4Ø FOR KK=JJ TO N1
9Ø5Ø IF ABS(A1(KK,JJ))>DD THEN DD=
     ABS(A1(KK,JJ)):LL=KK
9Ø6Ø NEXT KK
9Ø7Ø IF LL=JJ THEN 912Ø
9Ø8Ø FOR KK=JJ TO N1: REM
     INTERCHANGE EQUATIONS
9Ø9Ø DD= A1(JJ,KK): A1(JJ,KK)=A1(LL,
     KK): A1(LL,KK)=DD
91ØØ NEXT KK
911Ø DD=B1(JJ): B1(JJ)=B1(LL): B1(LL)=
     DD
912Ø FOR KK=JJ+1 TO N1: DD=A1(KK,
     JJ)/A1(JJ,JJ)
913Ø FOR LL=JJ TO N1 : REM ELIMINATE
914Ø A1(KK,LL)=A1(KK,LL)−DD★A1(JJ,
     LL)
915Ø NEXT LL
916Ø B1(KK)=B1(KK)−DD★B1(JJ)
917Ø NEXT KK
918Ø NEXT JJ
```

```
9190 FOR JJ=N1 TO 1 STEP -1: REM
       BACK SUBSTITUTE
9200 DD=B1(JJ)
9210 IF JJ=N1 THEN 9250
9220 FOR KK=JJ+1 TO N1
9230 DD=DD-X1(KK)*A1(JJ,KK)
9240 NEXT KK
9250 X1(JJ)=DD/A1(JJ,JJ)
9260 NEXT JJ
9270 RETURN
```

**Driver program:**

```
 10 INPUT "N1";N1
 20 DIM A1(N1,N1), B1(N1), X1(N1), Y1(N1)
 30 FOR J=1 TO N1
 40 FOR K=1 TO N1
 50 A1(J,K)= 100 *(RND(0)-0·5)
 60 NEXT K
 70 PRINT "Y1(";J; ")";: INPUT Y1(J)
 80 NEXT J
 90 FOR J=1 TO N1
100 B1(J)=0
110 FOR K=1 TO N1
120 B1(J)= B1(J)+Y1(K)*A1(J,K)
130 NEXT K,J
140 X=TI
150 GOSUB 9000
160 X=TI-X
170 FOR J=1 TO N1
180 PRINT Y1(J);X1(J)
190 NEXT J
200 PRINT "TIME ="; INT (X/600·5)
210 STOP
```

## NOTE ON DRIVER PROGRAM

This program is designed to exercise the subroutine for solving simultaneous equations and to present its results in a form which may be easily checked.

The program begins by asking the user for the number of equations to be solved. It then requests an appropriate number of values for the 'unknowns'.

Next, the program constructs a set of equations for the unknowns using random co-efficients. It solves them, and presents a set of results alongside the original values. The results should be the same, except for minor rounding errors.

*Sample runs:*

```
RUN
N1?  1
Y1(1)?  6
  6  6                      (Same!)
TIME =0
```

```
RUN
N1?  4
Y1(1)?  4
Y1(2)?  1
Y1(3)?  7
Y1(4)?  8
  4  4
  1  1                      Same
  7  7
  8  8·000000001  About the same
TIME =0
```

# APPENDIX C

## UNIT:16

### Experiment 16.1

```
  1Ø INPUT "WAGES IN CENTS";W
  2Ø FOR J=1 TO 8
  3Ø READ V,N$
  4Ø T=INT(W/V)
  5Ø PRINT T;N$
  6Ø W=W-V★T
  7Ø NEXTJ
  8Ø STOP
1ØØØ DATA 5ØØØ,FIFTY-DOLLAR BILLS
1Ø1Ø DATA 1ØØØ,TEN-DOLLAR BILL(S)
1Ø2Ø DATA 5ØØ,FIVE-DOLLAR BILL(S)
1Ø3Ø DATA 1ØØ,DOLLAR(S)
1Ø4Ø DATA 25,QUARTER(S)
1Ø5Ø DATA 1Ø,DIME(S)
1Ø6Ø DATA 5,NICKEL(S)
1Ø7Ø DATA 1,CENT(S)
```

### Experiment 16.2A

```
  1Ø FOR K=1 TO 12
  2Ø READ M$
  3Ø PRINT M$
  4Ø NEXT K
  5Ø STOP
1ØØØ DATA JANUARY,FEBRUARY,MARCH,
     APRIL
1Ø1Ø DATA MAY,JUNE,JULY,AUGUST
1Ø2Ø DATA SEPTEMBER,OCTOBER,NOVEMBER,
     DECEMBER
```

### Experiment 16.2B

```
  1Ø INPUT D,M,Y
  2Ø FOR J=1 TO M
  3Ø READ M$
  4Ø NEXT J
  5Ø PRINT D;M$;Y
  6Ø RESTORE
  7Ø GOTO 1Ø
1ØØØ DATA JANUARY,FEBRUARY,MARCH,
     APRIL
1Ø1Ø DATA MAY,JUNE,JULY,AUGUST
1Ø2Ø DATA SEPTEMBER,OCTOBER,NOVEMBER,
     DECEMBER
```

### Experiment 16.3

```
  1Ø T=Ø
  2Ø S=Ø
  3Ø PRINT" SHIFT and CLR HOME "
  4Ø READA$
  5Ø IF A$="END" THEN 24Ø
  6Ø READ B$
  7Ø T=T+1
  8Ø J=1
  9Ø PRINTA$
 1ØØ PRINT
 11Ø INPUT X$
 12Ø IF X$=B$ THEN 2ØØ
 13Ø IF J=3 THEN 17Ø
 14Ø J=J+1
 15Ø PRINT"WRONG. TRY AGAIN"
 16Ø GOTO 9Ø
 17Ø PRINT "THE ANSWER IS"
 18Ø PRINT B$
 19Ø GOTO 4Ø
 2ØØ PRINT "THAT'S RIGHT!"
 21Ø IF J>1 THEN 4Ø
 22Ø S=S+1
 23Ø GOTO 4Ø
 24Ø PRINT "YOU GOT";S;"RIGHT"
 25Ø PRINT "FIRST TIME"
 26Ø PRINT"OUT OF";T;"QUESTIONS"
 27Ø STOP
 28Ø DATAWHO COMPOSED THE MESSIAH,
     HANDEL
 29Ø DATA HOW MANY SYMPHONIES DID
     BEETHOVEN WRITE,NINE
 3ØØ DATA WHO WROTE THE OPERA CARMEN,
     BIZET
 31Ø DATA WHAT DID PAGANINI PLAY,VIOLIN
 32Ø DATA END
```

# UNIT:17

## Experiment 17.1A

```
10 INPUT"HOW MANY MINUTES";M:R=TI+M
   ★3600
20 IF TI<R THEN 20
30 PRINT"TIME UP":STOP
```

## Experiment 17.1B

```
10 PRINT"USE 1000000 TO":PRINT"END
   INPUT":S=0:N=0
20 INPUT"NEXT NUMBER";X:IF X=1000000
   THEN 40
30 S=S+X:N=N+1:GOTO 20
40 PRINT"AVERAGE =";S/N:STOP
```

## Experiment 17.1C

```
 10 REM DEBUG THIS PROGRAM!
 20 INPUT"NAME";N$
 30 IFN$="JIM"THEN A$="JAMES":GOTO
    100
 40 IFN$="BOB"THEN A$="ROBERT":GOTO
    100
 50 IFN$="KATE"THEN A$="KATHERINE":
    GOTO100
 60 IFN$="PENNY"THEN A$="PENELOPE":
    GOTO100
 70 PRINT N$;" IS NOT"
 80 PRINT"SHORT FOR ANYTHING."
 90 GOTO10
100 PRINT N$;" IS SHORT"
110 PRINT"FOR ";A$
120 GOTO10
```

## Experiment 17.2A

Most generous: SFF or SWWWW
NOT ((S < = 1 AND F < = 2 AND W = 0) OR
(S < = 1 AND F = 0 AND W < = 4)).
Most restrictive: SFF or WWWW
NOT ((S < = 1 AND F < = 2 AND W = 0) OR
(S = 0 AND F = 0 AND W < = 4)).

## Experiment 17.2B

N$<>"JONES" AND N$<>"SMITH" AND
N$<>"BROWN"
X>15 OR X<4

## Experiment 17.2C

```
 10 REM EXERCISE 17.2C
 20 IF T < 0.1 THEN PRINT "FANTASTIC!!":
    GOTO 100
 30 IF T < 0.15 THEN PRINT "AMAZINGLY
    GOOD!":GOTO 100
 40 IF T > = 0.15 AND T < 0.2 THEN PRINT
    "VERY GOOD":GOTO 100
 50 IF T > = 0.2 AND T < 0.25 THEN PRINT
    "GOOD":GOTO 100
 60 IF T > = 0.25 AND T < 0.28 THEN PRINT
    "FAIR":GOTO 100
 70 IF T > = 0.28 AND T < 0.33 THEN PRINT
    "PRETTY SLOW":GOTO 100
 80 IF T > = 0.33 AND T < 0.4 THEN PRINT
    "WAKE UP!":GOTO 100
 90 IF T > 0.4 THEN PRINT "TRY AGAIN WHEN
    YOU'RE SOBER!!"
100 STOP
```



## Experiment 17.2D

```
10 INPUT"WORD";W$
20 IF W$>="ABRAHAM" AND W$<=
   "FRANCE" THEN PRINT"USE VOL 1":STOP
30 IF W$>="FRANCHISE" AND W$<=
   LEVANT" THEN PRINT"USE VOL 2":STOP
40 IF W$>="LEVITATION" AND W$<=
   "QUOIT" THEN PRINT"USE VOL 3":STOP
50 IF W$>="QUOTIENT" AND W$<=
   "ZYLOPHONE" THEN PRINT "USE VOL 4":
   STOP
60 PRINT"THIS WORD IS NOT IN"
70 PRINT"THE ENCYCLOPAEDIA"
80 STOP
```

# UNIT:18

## Experiment 18.1A

```
 10 PRINT " SHIFT and CLR HOME ARITHMETIC
    TEST"
 20 PRINT "ANSWER THE FOLLOWING
    SUMS"
 30 S=0
 40 FOR A=1 TO 10
 50 X = INT (10★RND(0)+1)
 60 Y = INT (10★RND(0)+1)
 70 PRINT X; "+"; Y; "=";
 80 INPUT Z
 90 GOSUB 1000:REM MAKE SOUND
100 GOSUB 2000:REM CHANGE BORDER
    COLOUR
110 NEXT A
120 PRINT "THAT'S"; S; "RIGHT OUT OF 10"
130 FOR R=1 TO S
```

```
 140 GOSUB 1ØØØ:REM MAKE SOUND
 15Ø NEXT R
 16Ø STOP
1ØØØ REM SUBROUTINE TO MAKE PIP SOUND
1Ø1Ø POKE 36878, 15: POKE 36876, 245
1Ø2Ø FOR MM=1 TO 1ØØ:NEXT MM
1Ø3Ø POKE 36878,Ø
1Ø4Ø FOR MM=1 TO 8ØØ:NEXT MM
1Ø5Ø RETURN
2ØØØ REM SUBROUTINE TO CHANGE BORDER
     COLOUR
2Ø1Ø IF Z=X+Y THEN POKE 36879, 28: S=S+1:
     GOTO 2Ø3Ø:REM RIGHT ANSWER—
     BORDER PURPLE
2Ø2Ø POKE 36879,24: REM WRONG ANSWER
     — BORDER BLACK
2Ø3Ø FOR KK=1 TO 8ØØ:NEXT KK
2Ø4Ø POKE 36879, 27: REM RESTORE INITIAL
     COLOUR
2Ø5Ø RETURN
2Ø4Ø POKE 36879, 27: REM RESTORE INITIAL
     COLOUR
2Ø5Ø RETURN
```

## Experiment 18.1B

```
  1Ø PRINT" SHIFT and CLR HOME COUNTING
     TEST"
  2Ø PRINT"COUNT THE PIPS"
  3Ø S=Ø
  4Ø FOR A=1 TO 1Ø
  5Ø FOR T=1 TO 5ØØØ:NEXT T:REM WAIT A BIT
  6Ø X=INT(9★RND(Ø)+1)
  7Ø FOR J=1 TO X
  8Ø GOSUB 1ØØØ
  9Ø NEXT J
 1ØØ INPUT Z
 11Ø IF Z=X THEN GOSUB 5ØØØ:GOSUB 2ØØØ:
     GOSUB 4ØØØ:GOTO13Ø
 12Ø GOSUB3ØØØ:GOSUB 4ØØØ
 13Ø NEXT A
 14Ø PRINT"THAT'S";S;"RIGHT"
 15Ø STOP
1ØØØ REM MAKE PIP
1Ø1Ø POKE 36878,15:POKE 36876,245
1Ø2Ø FOR MM=1 TO 1ØØ:NEXT MM
1Ø3Ø POKE 36878,Ø
1Ø4Ø FOR MM=1 TO 8ØØ:NEXT MM
1Ø5Ø RETURN
2ØØØ REM SUBROUTINE TO CHANGE BORDER
     COLOUR PURPLE–RIGHT ANSWER
2Ø1Ø IFZ=XTHENPOKE36879,28:S=S+1:
     RETURN
3ØØØ REM SUBROUTINE TO CHANGE BORDER
     COLOUR BLACK–WRONG ANSWER
3Ø1Ø POKE36879,24:RETURN
4ØØØ REM SUBROUTINE TO RESTORE INITIAL
     COLOUR
4Ø1Ø FORKK=1TO8ØØ:NEXTKK
4Ø2Ø POKE 36879,27:RETURN
5ØØØ REM MYSTERY
5Ø1Ø POKE 36878, 15
5Ø2Ø FOR MM=15Ø TO 2ØØ STEP 1
5Ø3Ø POKE36876,MM
5Ø4Ø FOR TT=1 TO 1Ø:NEXT TT
5Ø5Ø NEXT MM
5Ø6Ø POKE 36878,Ø
5Ø7Ø RETURN
```

## Experiment 18.2

```
 1Ø PRINT" SHIFT and CLR HOME ";
 2Ø FOR X1=ØTO16
 3Ø C1$=" CTRL and RED "
 4Ø GOSUB 5ØØ
 5Ø FOR T=1TO15Ø :NEXT T
 6Ø C1$=" CTRL and WHT "
 7Ø GOSUB5ØØ
 8Ø NEXT X1
 9Ø GOTO 9Ø
5ØØ REM MONSTER
51Ø PRINT" CLR HOME  CRSR↑↓  CRSR↑↓ ";C1$;
52Ø IF X1=ØTHEN54Ø
53Ø FORJJ=1 TO X1:PRINT" CRSR⇐⇒ ";:NEXT JJ
54Ø PRINT" SPACE  SPACE  CTRL and
RVS ON  SHIFT and £  C= and *
CRSR↑↓  SHIFT and CRSR⇐⇒ twice  SPACE
CRSR↑↓  SHIFT and CRSR⇐⇒  SPACE
CRSR↑↓  SHIFT and CRSR⇐⇒ twice
SHIFT and £  SPACE  C= and *
CRSR↑↓  SHIFT and CRSR⇐⇒ 3 times
SPACE 3 times  CRSR↑↓  SHIFT and
CRSR⇐⇒ 3 times  CTRL and RVS OFF  C= and
*  CTRL and RVS ON  SPACE  CTRL
and RVS OFF  SHIFT and £  CRSR↑↓
SHIFT and CRSR⇐⇒ 3 times  SHIFT and
N  SPACE  SHIFT and M  CRSR↑↓
SHIFT and CRSR⇐⇒ 4 times  SHIFT and
V  SPACE 3 times  SHIFT and V ";
55Ø RETURN
```

## Experiment 18.3

```
10 PRINT" [SHIFT] and [CLR HOME] "
20 X1=1:Y1=22:N1=17:GOSUB2000
30 X1=1 :Y1=5:N1=3:GOSUB3000
40 X1=4:Y1=3:N1=3:GOSUB4000
50 X1=7:Y1=6:N1=5:GOSUB2000
60 X1=7:Y1=10:N1=13:GOSUB1000
70 X1=20:Y1=10:N1=11:GOSUB2000
80 X1=1:Y1=21:N1=20:GOSUB1000
90 X1=1:Y1=21:N1=20:GOSUB1000
100 X1=3:Y1=5:GOSUB5000
110 Y1=15
120 FORX1=2 TO 19 STEP 4
130 GOSUB6000
140 NEXT X1
150 X1=4:Y1=0:GOSUB7000:REM DRAW
    CROSS
160 GOTO160
500 REM POSITION CURSOR TO X1,Y1
510 PRINT"[CLR HOME] ";
520 IF X1=0 THEN 540
530 FORKK=1 TO X1:PRINT" [CRSR ⇒] ";:NEXT KK
540 IF Y1=0 THEN RETURN
550 FORKK=1 TO Y1:PRINT" [CRSR ⇓] ";:NEXT KK
560 RETURN
1000 REM TO DRAW HORIZONTAL LINE FOR
     N1 UNITS FROM X1,Y1
1010 GOSUB 500:REM POSITION CURSOR
1020 FOR JJ=1 TO N1
1030 PRINT" [CTRL] and [RVS ON] [SPACE] ";
1035 NEXT JJ
1037 PRINT" [CTRL] and [RVS OFF] ";
1040 RETURN
2000 REM DRAW VERTICAL LINE N1 UNITS
     DOWN FROM X1,Y1
2010 GOSUB 500:REM POSITION CURSOR
2020 FOR JJ=1 TO N1
2030 PRINT" [CTRL] and [RVS ON] [SPACE]
     [CRSR ⇓] [SHIFT] and [CRSR ⇒] ";
2040 NEXT JJ
2050 RETURN
3000 REM DRAW LINE DIAGONALLY UPWARDS
     AND RIGHT FROM X1,Y1
3010 GOSUB 500:REM POSITION CURSOR
3020 FOR KK=1 TO N1
3030 PRINT" [CTRL] and [RVS ON] [SHIFT]
     and [£] [CTRL] and [RVS OFF] [SHIFT]
     and [£] [SHIFT] and [CRSR ⇓] [SHIFT]
     and [CRSR ⇒] ";
3040 NEXT KK
3050 RETURN
4000 REM DRAW LINE DIAGONALLY DOWN-
     WARDS AND RIGHT FROM X1,Y1
4010 GOSUB 500:REM POSITION CURSOR
4020 FOR KK=1 TO N1
4030 PRINT" [C=] and [*] [CTRL] and
     [RVS ON] [C=] and [*] [CTRL] and
     [RVS OFF] [CRSR ⇓] [SHIFT] and [CRSR ⇒] ";
4040 NEXT KK
4050 RETURN
5000 REM DRAW WINDOW
5010 GOSUB 500:REM POSITION CURSOR
5020 PRINT" [C=] and [A] [SHIFT] and
     [*] [C=] and [S] [CRSR ⇓] [SHIFT] and
     [CRSR ⇒] 3 times [SHIFT] and [—] [SPACE]
     [SHIFT] and [—] [CRSR ⇓] [SHIFT] and
     [CRSR ⇒] 3 times [C=] and [Z] [SHIFT] and
     [*] [C=] and [X] ";
5030 RETURN
6000 REM PAINT ARCHED WINDOW
6010 GOSUB 500
6020 PRINT" [SHIFT] and [N] [SHIFT]
     and [M] [CRSR ⇓] [SHIFT] and [CRSR ⇒] twice
     [C=] and [+] twice [CRSR ⇓] [SHIFT] and
     [CRSR ⇒] twice [C=] and [+] twice [CRSR ⇓]
     [SHIFT] and [CRSR ⇒] twice [C=] and
     [+] twice ";
6030 RETURN
7000 REM DRAW CROSS
7010 GOSUB 500
7020 PRINT" [CTRL] and [RVS ON] [SPACE]
     [CRSR ⇓] [SHIFT] and [CRSR ⇒] twice
     [SPACE] 3 times [CRSR ⇓] [SHIFT] and
     [CRSR ⇒] twice [SPACE] [CRSR ⇓] [SHIFT] and
     [CRSR ⇒] [SPACE] ";
7030 RETURN
```

# UNIT:19

## Experiment 19.1

```
  1Ø INPUT"FIRST FRACTION";P,Q
  2Ø INPUT"SECOND FRACTION";S,T
  3Ø A1=P★T+Q★S
  4Ø B1=Q★T
  5Ø GOSUB 55ØØ
  6Ø PRINT "RESULT =";C1;"/";D1
  7Ø STOP
55ØØ REM REDUCE FRACTION R1/B1 TO ITS
       LOWEST TERMS
551Ø REM RESULT IN C1/D1 LOCALS ARE JJ,KK
552Ø JJ=A1:KK=B1
553Ø IF JJ=KK THEN 556Ø
554Ø IF JJ<KK THEN KK=KK-JJ: GOTO 553Ø
555Ø JJ=JJ-KK:GOTO 553Ø
556Ø C1=A1/JJ:D1=B1/JJ
557Ø RETURN
```

## Experiment 19.2B

```
  1Ø INPUT"FIRST FRACTION";P,Q
  2Ø INPUT"SECOND FRACTION";S,T
  3Ø A1=P★T+Q★S
  4Ø B1=Q★T
  5Ø GOSUB 55ØØ
  6Ø PRINT "RESULT =";C1;"/";D1
  7Ø STOP
55ØØ REM REDUCE FRACTION A1/B1 TO
       LOWEST TERMS, USING DIVISION NOT
       SUBTRACTION
551Ø REM RESULT IN C1/D1 LOCALS ARE
       JJ,KK,LL
552Ø REM ERROR IF A1 OR B1 NOT WHOLE
       NUMBERS OR IF B1<1
553Ø IF A1=INT(A1) AND B1=INT(B1)AND
       B1>=1 THEN 555Ø
554Ø PRINT"WRONG PARAMETERS—":PRINT
       A1;B1:STOP
555Ø IF A1=ØTHEN C1=Ø:D1=1:RETURN
556Ø LL=1:IFA1<ØTHEN LL=-1:A1=-A1
557Ø JJ=A1:KK=B1
558Ø IFKK=ØTHEN 562Ø
559Ø IFJJ=ØTHEN JJ=KK:GOTO562Ø
56ØØ IFJJ>KK THEN JJ=JJ-INT(JJ/KK)★KK:
       GOTO558Ø
561Ø KK=KK-INT(KK/JJ)★JJ:GOTO 558Ø
562Ø C1=LL★A1/JJ:D1=B1/JJ
563Ø RETURN
```

## Experiment 19.3

```
  1Ø INPUT"THREE NUMBERS";A1,B1,C1
  2Ø GOSUB 1ØØØ
  3Ø PRINT "LARGEST IS";X1
  4Ø GOTO 1Ø
1ØØØ REM FIND LARGEST OF A1,B1,C1
1Ø1Ø REM AND DELIVER RESULT IN X1
1Ø2Ø X1=A1
1Ø3Ø IF X1<B1 THEN X1=B1
1Ø4Ø IF X1<C1 THEN X1=C1
1Ø5Ø RETURN
```

## Experiment 19.4

```
  1Ø GET A1$:IFA1$=""THEN 1Ø
  2Ø GOSUB 8ØØØ
  3Ø GOTO 1Ø
8ØØØ REM DISPLAY CHARACTER IN A1$ FOUR
       TIMES USUAL SIZE
8Ø1Ø BB=ASC(A1$)
8Ø2Ø IFBB=13ORBB=141THENPRINT"
       CRSR 3 times ":RETURN
8Ø3Ø IFBB=18THENQQ=1:RETURN
8Ø4Ø IFBB=146THENQQ=Ø:RETURN
8Ø5Ø IFBB<32THENPRINTMID$(" CTRL and
RVS OFF 5 times CTRL and "2 CTRL
and RVS OFF 13 times CLR HOME CTRL and
RVS OFF 9 times CTRL and "2 CTRL
and RVS OFF CTRL and &6 CTRL and
'7 ",BB+1,1);:RETURN
8Ø6Ø IFBB>=144 AND BB<16ØTHENPRINT
       MID$(" CTRL and !1 CTRL
and RVS OFF twice SHIFT and CLR HOME
CTRL and RVS OFF 9 times CTRL and
%5 CTRL and RVS OFF CTRL and
(8 CTRL and $4 ",BB-143,1);:
       RETURN
8Ø7Ø AA=(BBAND31)+Ø.5★(BBAND128):IF
       (BBAND64)=ØTHENAA=AA+32
8Ø8Ø FORJJ=Ø TO 6 STEP 2
8Ø9Ø KK=PEEK(32768+8★AA+JJ):LL=PEEK
       (32769+8★AA+JJ)
81ØØ NN=64:FORMM=ØTO3
811Ø PP=1+8★INT(KK/NN)+2★INT(LL/NN)
812Ø KK=KK-INT(KK/NN)★NN:LL=LL-INT
       (LL/NN)★NN
813Ø IFQQ=ØTHENPRINTMID$(" CTRL
and RVS OFF SPACE CTRL and RVS OFF
C= and D CTRL and RVS OFF C=
and F CTRL and RVS OFF C= and
I CTRL and RVS OFF C= and C
CTRL and RVS ON C= and K
```

```
[CTRL] and [RVS ON] [C=] and [B]
[CTRL] and [RVS ON] [C=] and [V]
[CTRL] and [RVS OFF] [C=] and [V]
[CTRL] and [RVS OFF] [C=] and [B]
[CTRL] and [RVS OFF] [C=] and [K]
[CTRL] and [RVS ON] [C=] and [C]
[CTRL] and [RVS ON] [C=] and [I]
[CTRL] and [RVS ON] [C=] and [F]
[CTRL] and [RVS ON] [C=] and [D]
[CTRL] and [RVS ON] ",PP,2);:GOTO815Ø
814Ø PRINTMID$(" [CTRL] and [RVS ON]
[SPACE] [CTRL] and [RVS ON] [C=] and [D]
[CTRL] and [RVS ON] [C=] and [F]
[CTRL] and [RVS ON] [C=] and [I]
[CTRL] and [RVS OFF] [C=] and [C]
[CTRL] and [RVS OFF] [C=] and [K]
[CTRL] and [RVS OFF] [C=] and [B]
[CTRL] and [RVS OFF] [C=] and [V]
[CTRL] and [RVS ON] [C=] and [V]
[CTRL] and [RVS ON] [C=] and [B]
[CTRL] and [RVS ON] [C=] and [K]
[CTRL] and [RVS OFF] [C=] and [C]
[CTRL] and [RVS OFF] [C=] and [I]
[CTRL] and [RVS OFF] [C=] and [F]
[CTRL] and [RVS OFF] [C=] and [D]
[CTRL] and [RVS OFF] ",PP,2);
815Ø NN=INT(NN/4):NEXT MM
816Ø PRINT" [CRSR ⇅] [SHIFT] and [CRSR ⇔] 4 times";
817Ø NEXTJJ
818Ø PRINT" [SHIFT] and [CRSR ⇅] 4 times [CRSR ⇔]
      4 times";
819Ø IFPEEK(211)>18THENPRINT" [CRSR ⇅]
      3 times"
82ØØ RETURN
```

# UNIT: 20

## Experiment 20.1

```
1Ø DIM W$(1ØØ)
2Ø N=Ø
3Ø INPUT"NAME";X$
4Ø IF X$="ZZZZ" THEN 6Ø
5Ø N=N+1:W$(N)=X$:GOTO3Ø
6Ø FOR J=N TO 1 STEP -1
7Ø PRINT W$(J)
8Ø NEXT J
9Ø STOP
```

304

## Experiment 20.2A

```
  1Ø DIM T$(4Ø)
  2Ø FOR J=Ø TO 4Ø
  3Ø READ T$(J)
  4Ø NEXT J
  5Ø DATA NIL,I,II,III,IV,V,VI,VII,VIII,IX,X
  6Ø DATAXI,XII,XIII,XIV,XV,XVI,XVII,XVIII,XIX,
     XX
  7Ø DATAXXI,XXII,XXIII,XXIV,XXV,XXVI,XXVII,
     XXVIII,XXIX,XXX
  8Ø DATAXXXI,XXXII,XXXIII,XXXIV,XXXV,XXXVI,
     XXXVII,XXXVIII,XXXIX,XXXX
 1ØØ PRINT"GIVE TWO NUMBERS"
 11Ø INPUT X$,Y$
 12Ø A1$=X$:GOSUB 1ØØØ:X=B1
 13Ø A1$=Y$:GOSUB1ØØØ:Y=B1
 14Ø Z=X+Y
 15Ø IF Z>4Ø THEN PRINT"RESULT EXCEEDS
     CAPACITY":STOP
 16Ø PRINT"SUM = ";T$(Z)
 17Ø STOP
1ØØØ REM CONVERT WORD A1$ INTO ROMAN
     NUMBER B1
1Ø1Ø FOR JJ =Ø TO 4Ø
1Ø2Ø IF A1$=T$(JJ) THEN 1Ø5Ø
1Ø3Ø NEXT JJ
1Ø4Ø PRINT "NO ENTRY FOUND": STOP
1Ø5Ø B1=JJ
1Ø6Ø RETURN
```

## Experiment 20.2B1

```
  7 REM SOLUTION WITH ARRAYS
 1Ø DIM N$(2Ø),T$(2Ø)
 2Ø FORJ=1 TO 2Ø
 3Ø READ N$(J),T$(J)
 4Ø NEXTJ
 5Ø INPUT "NAME";X$
 6Ø FOR J=1 TO 2Ø
 7Ø IF X$=N$(J) THEN PRINT X$;"'S PHONE IS
    ";T$(J):PRINT:GOTO 5Ø
 8Ø NEXT J
 9Ø PRINTX$;" HAS NO LISTED"
```

```
 100 PRINT"PHONE NUMBER"
 110 GOTO 50
1000 DATA MAXWELL,3398123
1010 DATABOHR,558
1020 DATAEINSTEIN,4073189
1030 DATAVON NEUMANN,777000
1040 DATANEWTON,3074
1050 DATAZUSE,222
1060 DATAPLANCK,1237543
1070 DATABOYLE,146543
1080 DATABABBAGE,03474
1090 DATALAPLACE,5674
1100 DATAPTOLEMY,54863
1110 DATAARISTOTLE,66543
1120 DATAMCCARTHY,47
1130 DATADIJKSTRA,645
1140 DATABERZELIUS,777
1150 DATACHARLES,5543
1160 DATAMENDELEEV,645634
1170 DATATSIOLKOVSKY,645332
1180 DATAARCHIMEDES,2
1190 DATAHOYLE,21352
```

## Experiment 20.2B2

```
   7 REM SOLUTION WITHOUT ARRAYS
  10 RESTORE
  20 INPUT"NAME";X$
  30 FORJ=1 TO 20
  40 READ N$,T$
  50 IF N$=X$THEN PRINTX$;"'S PHONE IS
     ";T$:PRINT:GOTO 10
  60 NEXT J
  70 PRINTX$;" HAS NO LISTED"
  80 PRINT"PHONE NUMBER"
  90 GOTO 10
1000 DATA MAXWELL,3398123
1010 DATABOHR,558
1020 DATAEINSTEIN,4073189
1030 DATAVON NEUMANN,777000
1040 DATANEWTON,3074
1050 DATAZUSE,222
1060 DATAPLANCK,1237543
1070 DATABOYLE,146543
1080 DATABABBAGE,03474
1090 DATALAPLACE,5674
1100 DATAPTOLEMY,54863
1110 DATAARISTOTLE,66543
1120 DATAMCCARTHY,47
1130 DATADIJKSTRA,645
1140 DATABERZELIUS,777
1150 DATACHARLES,5543
1160 DATAMENDELEEV,645634
1170 DATATSIOLKOVSKY,645332
1180 DATAARCHIMEDES,2
1190 DATAHOYLE,21352
```

# UNIT: 21

## Experiment 21.1A

```
10 INPUT"TYPE A STRING";X$
20 Y$=""
30 FOR J=1 TO LEN(X$)
40 IF MID$(X$,J,1)="E" THEN 60
50 Y$=Y$+MID$(X$,J,1):GOTO 70
60 Y$=Y$+"O"
70 NEXT J
80 PRINT Y$
90 STOP
```

## Experiment 21.1B

```
10 PRINT" [SHIFT] and [CLR HOME] "
20 PRINT" [CLR HOME] [CRSR ⇑⇓] 6 times [CRSR ⇐⇒] 7 times";
30 PRINTMID$(TI$,1,2);"/";MID$(TI$,3,2);"/";
   MID$(TI$,5,2)
40 GOTO20
```

## Experiment 21.1C

```
  10 INPUT "NAME PLEASE";N1$
  20 GOSUB 4100
  30 PRINT "SURNAME IS ";Y1$
  40 GOTO10
4100 REM EXTRACT SURNAME FROM N1$ AND
     DELIVER IT IN Y1$ (IMPROVED VERSION)
4110 JJ=LEN(N1$)
4120 IF JJ=0 THEN Y1$="":RETURN
4130 IF MID$(N1$,JJ,1)<"A" OR MID$
      (N1$,JJ,1)>"Z" THEN JJ=JJ-1:GOTO
      4120
4140 FORKK=JJ TO 1 STEP-1
4150 CC$=MID$(N1$,KK,1)
4160 IF NOT(CC$>="A" AND CC$<="Z" OR
     CC$="-" OR CC$="")THEN 4190
4170 NEXT KK
4180 KK=0
4190 Y1$=MID$(N1$,KK+1,JJ-KK)
4200 RETURN
```

## Experiment 21.2

```
  10 FOR J=667TO 677
  20 FOR Y1=0TO 3
  30 X1=SQR(J)
  40 GOSUB5000
  50 NEXT Y1
  60 PRINT
  70 NEXT J
  80 STOP
5000 REM DISPLAY X1 TO Y1 DECIMAL PLACES
5010 IF Y1>0 AND ABS(X1)<=999999999 THEN
```

```
        GOTO 5050
5020 X1=X1+0.5
5030 PRINT INT(X1);
5040 RETURN
5050 IF X1<0THEN X1=X1-0.5★10↑-Y1:
        GOTO 5070
5060 X1=X1+0.5★10↑-Y1
5070 NN$=STR$(X1)
5080 FOR PP = 1 TO LEN(NN$)
5090 IF MID$(NN$,PP,1)="." THEN PRINT
        LEFT$(NN$,PP+Y1);:RETURN
5100 NEXT PP
5110 PRINT NN$; ".";
5120 FORJJ=1TO Y1:PRINT"0";:NEXTJJ
5130 RETURN
```

## Experiment 21.3A

```
10 INPUT"STRING";N$
20 FORJ=1 TO LEN(N$)
30 IF MID$(N$,J,1)>="A" AND MID$(N$,J,1)
    <="Z" THEN 60
40 NEXT J
50 PRINT "NO WORD":STOP
60 N=VAL(LEFT$(N$,J-1))
70 N$=RIGHT$(N$,LEN(N$)-J+1)
80 PRINT N$, 2★N
90 GOTO 10
```

## Experiment 21.3B

```
  10 DIM N1$(10),Q1(10)
  20 INPUT"LIST";X1$
  30 GOSUB 8000
  40 FORJ=1TOX
  50 PRINT N1$(J),Q1(J)
  60 NEXT J
  70 STOP
8000 REM PARSE SHOPPING LIST IN X1$
8010 X=0:JJ=1:LL=LEN(X1$)
8020 GOSUB 8200:REM FIRST LOOK FOR THE
        FIRST DIGIT OF A NUMBER
8030 IF JJ>LL THEN RETURN :REM OUT IF
        STRING ENDED
8040 GOSUB 8300:REM EXTRACT NUMBER
        (DELIVERED IN NN)
8050 IF JJ>LL THEN 8100:REM SIGNAL FAULT
        IF STRING ENDED
8060 X=X+1:Q1(X)=NN:REM PUT NUMBER
        AWAY
8070 GOSUB 8400:REM EXTRACT WORD IN
        S1$.Z1=0 IF WORD CAN'T BE FOUND
8080 IF Z1=0THEN 8100
8090 N1$(X)=S1$:GOTO 8020
8100 PRINT"DON'T UNDERSTAND":X=0:
        RETURN
8200 REM LOOK FOR START OF NUMBER IN
        X1$
8210 IF JJ>LL THEN RETURN
8220 CC$=MID$(X1$,JJ,1)
8230 IFCC$<"0" OR CC$>"9"THEN JJ=JJ+1:
        GOTO8210
8240  RETURN
8300 REM EXTRACT NUMBER FROM X1$
        STARTING AT JJ, AND DELIVER IN NN.
        ADVANCE
8310 KK=JJ:JJ=JJ+1
8320 IFJJ>LLTHEN RETURN
8330 CC$=MID$(X1$,JJ,1)
8340 IF CC$>="0" AND CC$<="9" THEN
        JJ=JJ+1:GOTO8320
8350 NN=VAL(MID$(X1$,KK,JJ-KK)):RETURN
8400 REM EXTRACT WORD FROM X1$
        STARTING AT JJ.DELIVER IN S1$,AND
        ADVANCE JJ
8405 REM Z1=0 IF NO WORD CAN BE FOUND
8410 IFJJ>LL THEN Z1=0:RETURN
8420 CC$=MID$(X1$,JJ,1)
8430 IF CC$<"A"OR CC$>"Z" THEN JJ=JJ+1:
        GOTO 8410
8440 KK=JJ:JJ=JJ+1
8450 IF JJ>LLTHEN 8480
8460 CC$=MID$(X1$,JJ,1)
8470 IFCC$>="A" AND CC$<="Z"THEN JJ=
        JJ+1:GOTO8450
8480 S1$=MID$(X1$,KK,JJ-KK):Z1=1:RETURN
```



# UNIT: 22

## Experiment 22.1

```
  10 DATABAIN,BEAVIS,BOWEY,BURNS,
       CLARK,FLEMING
  20 DATAGORDON,GREEN,HOOD,KIDD,
       MCCABE,MAVER
  30 DATAMARSHALL,MILLER,NORTH,PACK,
       PERKINS,REED,ROSE
  40 DATAROSS,SIMPSON,SMITH,SYKES,
       TEDFORD,WEBSTER,WOOD
  50 DIMA$(26)
  60 FORJ=1 TO 26:READA$(J):NEXT J
  70 INPUT"TYPE A NAME";X$
  80 H1=26:L1=1:GOSUB6000
  90 IFM1=-1THEN PRINT X$;" NOT
       FOUND":GOTO70
 100 PRINT X$;" FOUND AT ENTRY";M1
 110 GOTO70
6000 REM SEARCH ORDERED :LIST A$
6010 IF H1<L1 THEN M1=-1:RETURN
6020 M1=INT(0.5★(H1+L1))
6030 IF X$=A$(M1)THEN RETURN
6040 IF X$<A$(M1)THEN
       H1=M1-1:GOTO6010
6050 L1=M1+1:GOTO6010
```

## Experiment 22.2

```
  10 DATAROSS,SIMPSON,SMITH,SYKES,
       TEDFORD,WEBSTER,WOOD
  20 DATAMARSHALL,MILLER,NORTH,PACK,
       PERKINS,REED,ROSE
  30 DATAGORDON,GREEN,HOOD,KIDD,
       MCCABE,MAVER
```

```
  4Ø DATABAIN,BEAVIS,BOWEY,BURNS,
     CLARK,FLEMING
  5Ø DIMA1$(26)
  6Ø FORJ=1 TO 26:READA1$(J):NEXT J
  7Ø N1=26:GOSUB65ØØ
  8Ø FOR J=1 TO 26
  9Ø PRINT A1$(J)
 1ØØ NEXT J
 11Ø STOP
65ØØ REM BUBBLE SORT N1 ITEMS IN A1$(1)TO
     A1$(N1)
651Ø SS$="NO"
652Ø FOR KK=1 TO N1-1
653Ø IFA1$(KK)>A1$(KK+1)THENDD$=A1$
     (KK):A1$(KK)=A1$(KK+1):A1$(KK+1)=
     DD$:SS$="YES"
654Ø NEXT KK
655Ø IF SS$="YES"THEN 651Ø
656Ø RETURN
```

## Experiment 22.3A

```
 1Ø DATAADAMS,27
 2Ø DATABRIGGS,66
 3Ø DATACHILVERS,29
 4Ø DATADALE,38
 5Ø DATACOLIN,12
 6Ø DATAMACSNOOT,67
 7Ø DATAWILSON,96
 8Ø DATA THOMSON,53
 9Ø DATAWILMOTT,31
1ØØ DATABAIN,42
11Ø DATAMUNDY,64
12Ø DATAKRESTIN,85
13Ø DATAMCILDOWIE,1Ø
14Ø DATAWRAITH,99
15Ø DATAGREEN,72
16Ø DATAGREENE,52
17Ø DATASHEPHERD,53
18Ø DATAHUTCHISON,64
19Ø DATABLACK,45
2ØØ DATABAXTER,1
21Ø DATASMYRL,99
22Ø DATAFLASHMAN,2
23Ø DATAMORRIS,75
24Ø DATAELLIS,42
25Ø DATAMATTHEWS,66
26Ø DATAFOLEY,91
27Ø DATACOLLINS,36
28Ø DATADANIELS,93
29Ø DATAJACKSON,77
3ØØ DATAPEIRCE,78
31Ø DATADEWEY,34
32Ø DATAMACGREGOR,15
33Ø DATAEASON,69
34Ø DATAPARSONS,6
35Ø DATATHATCHER,45
36Ø DATACLAYMORE,85
37Ø DATAO'FLAHERTY,66
38Ø DATABUNN,5
39Ø DATASULLIVAN,85
4ØØ DATAGILBERT,41
5ØØ N1=4Ø:REM NUMBER OF PUPILS
51Ø DIM S$(N1),M(N1),A1(N1)
52Ø FOR J=1TON1
53Ø READ S$(J),M(J)
54Ø A1(J)=M(J)
55Ø NEXT J
56Ø REM NOW SORT MARKS IN A1
57Ø GOSUB 6ØØØ
58Ø REM FIND PASS-MARK (THREE QUARTERS-
    WAY DOWN LIST)
59Ø P=A1(INT(Ø.75*N1+1))
6ØØ PRINT"PASS-MARK IS";P
61Ø REM NOW DISPLAY STUDENTS WHO PASS
62Ø FOR J=1 TO N1
63Ø IF M(J)>=P THEN PRINT S$(J),M(J)
64Ø NEXT J
65Ø STOP
6ØØØ REM QUICKSORT:SORTS N1 ELEMENTS
     OF A1
6Ø1Ø IFSS=1THEN6Ø3Ø
6Ø2Ø DIMSS%(1ØØ):SS=1:REM DECLARE STACK
6Ø3Ø AA=1:BB=N1:SS%(Ø)=1:PP=1
6Ø4Ø XX=AA:YY=BB:ZZ=A1(BB)
6Ø5Ø IFXX>=YYTHEN6Ø9Ø
6Ø6Ø IFA1(XX)<=ZZTHENXX=XX+1:GOTO
     6Ø5Ø
6Ø7Ø IFA1(YY)>=ZZTHENYY=YY-1:GOTO
     6Ø5Ø
6Ø8Ø DD=A1(YY):A1(YY)=A1(XX):A1(XX)=
     DD:GOTO6Ø5Ø
6Ø9Ø A1(BB)=A1(XX):A1(XX)=ZZ
61ØØ IFXX-AA<=1THEN614Ø
611Ø SS%(PP)=XX:SS%(PP+1)=BB:SS%(PP+2)
     =2:PP=PP+3
612Ø BB=XX-1:GOTO6Ø4Ø
613Ø PP=PP-3:XX=SS%(PP):BB=SS%(PP+1)
614Ø IFBB-XX<=1THEN617Ø
615Ø SS%(PP)=3:PP=PP+1:AA=XX+1:GOTO
     6Ø4Ø
616Ø PP=PP-1
617Ø ONSS%(PP-1)GOTO618Ø,613Ø,616Ø
618Ø RETURN
```

## Experiment 22.3B

```
 1Ø REM FIRST PART OF LIFE PROGRAM
 2Ø DIMX$(9,9),Y$(9,9)
 3Ø PRINT" SHIFT and CLR HOME TYPE
    STARTING PATTERN"
 4Ø PRINT"IN *'S AND SPACES,"
 5Ø PRINT"USING CURSOR CONTROLS"
 6Ø PRINT"TO GET IT INSIDE THE"
 7Ø PRINT"BOX. USE RETURN"
 73 PRINT "TO FINISH EACH ROW"
 77 PRINT"(EVEN EMPTY ONES)."
 8Ø PRINT" 3 spaces C= and A
    SHIFT and * 9 times C= and
    S "
 9Ø FORJ=1 TO 9
1ØØ PRINT" 3 spaces SHIFT and -
    9 spaces SHIFT and - "
```

```
11Ø NEXTJ
12Ø PRINT"  3 spaces [C=] and [Z]
     [SHIFT] and [*] 9 times [C=] and
     [X] "
13Ø PRINT" [CLR HOME] [CRSR ↑↓] 8 times";
14Ø FOR J=1 TO 9
15Ø T$="":INPUT T$
16Ø FORK=2TO 1Ø
17Ø X$(J,K−1)=MID$(T$,K,1)
18Ø NEXTK,J
19Ø REM DISPLAY CURRENT POSITION
2ØØ PRINT" [SHIFT] and [CLR HOME] [CRSR ↑↓] 3 times
     3 spaces [C=] and [A] [SHIFT]
     and [*] 9 times [C=] and [S] "
21Ø FORJ=1 TO 9
22Ø PRINT"  3 spaces [SHIFT] and [−] ";
23Ø FORK=1TO9:PRINTX$(J,K);:NEXTK
24Ø PRINT" [SHIFT] and [−] "
25Ø NEXTJ
26Ø PRINT"  3 spaces [C=] and [Z]
     [SHIFT] and [*] 9 times [C=] and
     [X] "
27Ø FOR J=2 TO 8
28Ø FOR K=2 TO 8
29Ø T=Ø
3ØØ IFX$(J−1,K−1)="★"THEN T=T+1
31Ø IFX$(J−1,K)="★"THEN T=T+1
32Ø IFX$(J−1,K+1)="★"THEN T=T+1
33Ø IFX$(J,K−1)="★"THEN T=T+1
34Ø IFX$(J,K+1)="★"THEN T=T+1
35Ø IFX$(J+1,K−1)="★"THEN T=T+1
36Ø IFX$(J+1,K)="★"THEN T=T+1
37Ø IFX$(J+1,K+1)="★"THEN T=T+1
38Ø Y$(J,K)=X$(J,K)
39Ø IF T<2 OR T>3THEN Y$(J,K)="  "
4ØØ IF T=3THENY$(J,K)="★"
41Ø NEXT K,J
42Ø FORJ=2 TO 8
43Ø FORK=2 TO 8
44Ø X$(J,K)=Y$(J,K)
45Ø NEXT K,J
46Ø GOTO19Ø
```

# UNIT:23

## Experiment 23.1B

```
  1Ø INPUT"PEEK ADDRESS";J
  2Ø FOR K=Ø TO 7
  3Ø X1=PEEK(J+K)
  4Ø GOSUB 1ØØØ
  5Ø NEXT K
  6Ø PRINT:PRINT
  7Ø GOTO1Ø
1ØØØ REM GIVEN A LOCATION IN X1, WORK
      OUT THE CORRESPONDING BINARY
      PATTERN IN IT
1Ø1Ø YY=256:FOR KK=1TO 8
1Ø2Ø YY=YY/2
1Ø3Ø IFX1>=YY THEN X1=X1−YY:PRINT"★";:
      GOTO1Ø5Ø
1Ø4Ø PRINT" ";
1Ø5Ø NEXTKK
1Ø6Ø PRINT:RETURN
```

# UNIT:24

## Experiment 24.1

```
 1Ø PRINT" [SHIFT] and [CLR HOME] ";
 2Ø X=11:Y=11
 3Ø POKE 768Ø+22★Y+X,16Ø
 4Ø POKE 384ØØ+22★Y+X,Ø
 5Ø GET A$:IF A$="" THEN5Ø
 6Ø IF ASC(A$)<> 133 THEN 8Ø
 7Ø IF Y>ØTHEN Y=Y−1:GOTO3Ø
 8Ø IFASC(A$)<>134 THEN 1ØØ
 9Ø IF X<21 THEN X=X+1:GOTO3Ø
1ØØ IF ASC(A$)<>135THEN12Ø
11Ø IF Y<22THEN Y=Y+1:GOTO3Ø
12Ø IF ASC(A$)<>136THEN GOTO5Ø
13Ø IF X>ØTHEN X=X−1:GOTO 3Ø
14Ø GOTO5Ø
```

## Experiment 24.2A

```
 1Ø DEF FNA(X)=X↑3+(X+7)↑2−1ØØ
 5Ø FOR J=2TO3STEPØ.1
 6Ø PRINT J;FNA(J)
 7Ø NEXT J
 8Ø STOP
1ØØ REM BEST ESTIMATE FOR SOLUTION IS
     ABOUT 2.33
```

**Experiment 24.2B**

```
1Ø DEF FNA(X)=X↑3+(X+7)↑2–1ØØ
2Ø DEF FNB(X)= 3★X↑2 +2★(X+7)
3Ø X=2
4Ø Y=FNA(X)
5Ø PRINT "X=";X
6Ø PRINT"Y=";Y
7Ø IF ABS(Y)>Ø.ØØØØØØ1 THEN X=X–Y/FNB(X):
   GOTO4Ø
8Ø PRINT"SOLUTION IS";X
9Ø STOP
```

**Experiment 24.3A**

```
1Ø INPUT"FILE NAME";F$
2Ø OPEN1,1,2,F$
3Ø PRINT
4Ø PRINT"TYPE A SENTENCE AND END IT
   WITH A FULL STOP."
5Ø PRINT"DON'T EXCEED 75 CHARACTERS"
6Ø PRINT"YOUR LAST SENTENCE SHOULD BE
   ZZ."
7Ø S$=" "
8Ø GETA$:IFA$=" "THEN 8Ø
9Ø IF ASC(A$)=2ØTHENPRINT" [SHIFT] and
   [CRSR ⇐⇒] ";:GOTO8Ø
1ØØ PRINTA$;:S$=S$+A$
11Ø IF A$<>"." AND LEN(S$)<=75 THEN 8Ø
12Ø PRINT#1,S$
13Ø IFLEFT$(S$,2) <>"ZZ"THEN 3Ø
14Ø CLOSE1
15Ø STOP
```

**Experiment 24.3B**

```
1Ø DIM J(26,5)
2Ø FORQ=1TO26
3Ø FORR=1TO5
4Ø READJ(Q,R)
5Ø NEXTR,Q
1ØØ DATA1,3,Ø,Ø,Ø
11Ø DATA3,1,1,1,Ø
12Ø DATA3,1,3,1,Ø
13Ø DATA3,1,1,Ø,Ø
14Ø DATA1,Ø,Ø,Ø,Ø
15Ø DATA1,1,3,1,Ø
16Ø DATA3,3,1,Ø,Ø
17Ø DATA1,1,1,1,Ø
18Ø DATA1,1,Ø,Ø,Ø
19Ø DATA1,3,3,3,Ø
2ØØ DATA3,1,3,Ø,Ø
21Ø DATA1,3,1,1,Ø
22Ø DATA3,3,Ø,Ø,Ø
23Ø DATA3,1,Ø,Ø,Ø
24Ø DATA3,3,3,Ø,Ø
25Ø DATA1,3,3,1,Ø
26Ø DATA3,3,1,3,Ø
27Ø DATA1,3,1,Ø,Ø
28Ø DATA1,1,1,Ø,Ø
29Ø DATA3,Ø,Ø,Ø,Ø
3ØØ DATA1,1,3,Ø,Ø
31Ø DATA1,1,1,3,Ø
32Ø DATA1,3,3,Ø,Ø
33Ø DATA3,1,1,3,Ø
34Ø DATA3,1,3,3,Ø
35Ø DATA3,3,1,1,Ø
5ØØ INPUT"FILE NAME";Q$
51Ø OPEN 1,1,Ø,Q$
52Ø INPUT "RATE?";R
53Ø INPUT#1,Z$
535 IF LEN(Z$)=ØTHEN53Ø
54Ø IF LEFT$(Z$,2)="ZZ" THEN 65Ø
55Ø IF ST<>Ø THEN 8ØØ
56Ø POKE36876,24Ø
57Ø FOR W=1TOLEN(Z$)
58Ø A1$=MID$(Z$,W,1)
59Ø GOSUB2ØØØ
6ØØ NEXTW
61Ø GOTO 53Ø
65Ø PRINT "END OF TEXT"
66Ø STOP
8ØØ PRINT"TAPE ERROR"
81Ø STOP
1ØØØ REM MAKE PIP LENGTH T1 VOL V1
1Ø1Ø POKE 36878,V1
1Ø2Ø FOR TT=1 TO T1:NEXTTT
1Ø3Ø RETURN
2ØØØ REM SEND CHARACTER IN A1$
2Ø1Ø AA=ASC(A1$)
2Ø2Ø IF AA=32 THEN T1=4★R:GOSUB1ØØØ:
     RETURN
2Ø3Ø IFAA<65ORAA>9ØTHEN RETURN
2Ø4Ø AA=AA–64:PP=1
2Ø5Ø IF J(AA,PP)=Ø THEN T1=2★R:GOSUB
     1ØØØ:RETURN
2Ø6Ø T1=J(AA,PP)★R:V1=15:GOSUB 1ØØØ
2Ø7Ø T1=R:V1=Ø:GOSUB1ØØØ:PP=PP+1:
     GOTO2Ø5Ø
```

**Experiment 24.4**

```
1Ø DATAEDINBURGH,GLASGOW,DUNDEE,
   ABERDEEN
2Ø DATAFOOTBALL,TENNIS,HILLWALKING,
   OPERA,JAZZ,ROCK,THEATRE,READING,
   POLITICS
3Ø DATACHESS,GAMBLING,HORSERACING,
   CARS,MOTORBIKES,CYCLING,MEETING
   PEOPLE
4Ø DATACONSERVATIVE,LABOUR,LIBERAL,
   SDP,OTHER,NONE
6Ø DIMP$(26)
7Ø FORJ=1TO26:READP$(J):NEXT
8Ø PRINT" [SHIFT] and [CLR HOME] COMPUTER
   DATING"
9Ø PRINT"WHAT TOWN?"
1ØØ A1=1:B1=4:GOSUB1ØØØ:XT$=P1$
11Ø PRINT" [SHIFT] and [CLR HOME] WHAT IS
    YOUR MAIN"
12Ø PRINT"INTEREST?"
13Ø A1=5:B1=2Ø:GOSUB1ØØØ:XH$=P1$
```

```
140 PRINT" SHIFT and CLR HOME WHAT IS
     YOUR SECOND"
150 PRINT"INTEREST?"
160 A1=5:B1=2Ø:GOSUB1ØØØ:X1$=P1$
170 PRINT" SHIFT and CLR HOME WHAT ARE
     YOUR"
180 PRINT"POLITICS?"
190 A1=21:B1=26:GOSUB1ØØØ:XP$=P1$
200 PRINT"ARE YOU MALE OR FE-"
210 PRINT"MALE (SAY M OR F)
220 INPUTXS$
230 IF XS$<>"M" AND XS$<>"F"THEN 2ØØ
240 INPUT"AGE";XA
250 INPUT"HEIGHT IN INCHES";XH
260 OPEN1,1,Ø,"COMPUTER DATES"
265 M=-1ØØ:PRINT" SHIFT and CLR HOME ";
270 INPUT#1,C$,A$,T$,S$,A,H,H1$,H2$,PO$
280 IFST<>Ø THEN 7ØØ
290 IFT$<>XT$ORS$=XS$THEN 27Ø
320 S=Ø
330 D=XA-A:IFXS$="F"THEND=-D:REM
     D=MALE'S AGE — FEMALE'S AGE
340 IFD>=Ø AND D<=4 THEN S=S+5
350 D=XH-H:IFXS$="F"THEND=-D
360 IFD>=1 AND D<=3 THEN S=S+3
370 IFXH$=H1$ORXH$=H2$THENS=S+6
380 IFXI$=H1$ORXI$=H2$THENS=S+6
390 IFXP$=PØ$THENS=S+4
400 IFXP$="CONSERVATIVE"ANDPO$=
     "LABOUR" THEN S=S-2
410 IFPO$="CONSERVATIVE"ANDXP$=
     "LABOUR" THEN S=S-2
430 IFS>=MTHENM=S:PRINT" SHIFT and
     CLR HOME BEST SOLUTION SO FAR":GOSUB
     11ØØ:PRINT"SCORE=";S
440 GOTO27Ø
700 STOP
999 STOP
1000 REM DISPLAY MENU A1 TO B1 OF P$.
     SELECT A WORD AND RETURN IN P1$
1010 JJ=1
1020 FORKK=A1TOB1
1030 PRINTJJ;">";P$(KK):JJ=JJ+1
1040 NEXTKK
1045 PRINT
1050 INPUT"CHOOSE A NUMBER";LL:LL=INT
     (LL)
1060 IF LL<1 OR LL>B1-A1+1THEN 1Ø1Ø
1070 P1$=P$(LL+A1-1):RETURN
1100 REM DISPLAY PERSON
1110 PRINT"NAME:";C$
1120 PRINT"ADDRESS:";A$
1130 PRINT"TOWN:";T$
1140 PRINT"AGE:";A
1150 PRINT"HEIGHT:";H
1160 PRINT"HOBBIES:";H1$
1170 PRINT"         ";H2$
1180 PRINT"POLITICS:";PO$
1190 RETURN
2000 OPEN1,1,Ø
2010 GET#1,A$
2020 PRINTA$
2030 GOTO2Ø1Ø
```

310

# INDEX

**Commodore International Limited**
487 Devon Park Drive, Suite 200,
Wayne, PA 19087

PRINTED IN ENGLAND

commodore
COMPUTER